Generating Processors from Specifications of Instruction Sets

Dissertation

A thesis submitted to the
Faculty of Electrical Engineering, Computer Science, and Mathematics
of the
University of Paderborn

in partial fulfillment of the requirements for the degree of
Doctor rerum naturalium (Dr. rer. nat.)

by

Ralf Dreesen

Paderborn, September 2011

Supervisor:

Prof. Dr. Uwe Kastens (University of Paderborn)

Reviewer:

Prof. Dr. Uwe Kastens (University of Paderborn)
Prof. Dr. Marco Platzner (University of Paderborn)
Prof. Dr.-Ing. Ulrich Rückert (University of Bielefeld)

Additional members of committee:

Prof. Dr. Franz Rammig (University of Paderborn)
Dr. Matthias Fischer (University of Paderborn)

Submitted: 15.09.2011
Examination: 6.12.2011
Published: 9.12.2011

Acknowledgments

I would like to use this opportunity to show my gratitude to all people who supported me during this thesis.

First of all, I would like to thank my adviser Professor Uwe Kastens, for the support and guidance he gave me and which significantly contributed to the quality of this thesis. Professor Kastens allowed me great latitude in the design of the system and has shown me the right direction during numerous fruitful discussions. I want to thank Professor Ulrich Rückert and Professor Marco Platzner for their interest in my thesis and reviewing. In addition, I would like to thank Professor Rückert for the constructive cooperation with his research group.

I am obliged to many of my colleagues, who accompanied me in the past years. In particular, I owe earnest thankfulness to Dr. Michael Thies for many intensive discussions during coffee break. Special thanks go to Thorsten Jungeblut for helping me in the area of hardware design and giving me access to the respective infrastructure.

I would like to show my gratitude to Friederike Sudholt and Paul Jaessing for proof reading, in particular as the subject is far away from their profession. My personal thanks go to Johanna Sudholt, who has supported me throughout this thesis, as she has done for the past 12 years. I am thankful to my parents for giving me advice and virtues for my life.

Paderborn, 15.09.2011 Ralf Dreesen

Abstract

Most digital systems include microprocessors, as they are very flexible. Some of these microprocessors are tailored to the respective area of application, to optimize execution time and power consumption. An efficient development of such processors necessitates convenient development tools.

This thesis contributes a *specification language* called ViDL and two *generators* for rapid development of application specific processors. A developer specifies an instruction set in ViDL, then generates an instruction set simulator as well as a microarchitectural processor implementation from that specification. ViDL provides powerful concepts to specify a wide range of instruction sets at a high level of abstraction. For instance, the pipeline and its control are not defined by the developer, but contributed automatically by the processor generator. To demonstrate the power of ViDL, real world instruction sets, such as MIPS, ARM, Power and CoreVA have been specified. A ViDL specification is very similar to instruction set manuals, which allows for rapid formalization (e.g. one day for MIPS).

The generated high speed simulators execute 60 million instructions per second (Mips) on average with a peak performance of 140 Mips. The generated processor implementations yield a clock frequency of 600 MHz on a 65nm ST Microelectronics low power technology. The processor generator produces a pipelined n-stage microarchitecture, where n is automatically derived from a user defined target clock frequency (e.g. 600 MHz) and instruction semantics. As a result, different microarchitectural implementations (e.g. 2-stage and 6-stage) can be generated at no extra effort. All processors and the simulator are guaranteed to be consistent, as they are generated from the very same specification. There is no way that a ViDL user can break this consistency, neither by mistake nor by intention.

To prove the fitness of ViDL for *design space exploration (DSE)* and *instruction set extension (ISE)*, a new application specific processor has been developed as part of this thesis. The processor called DNACore is based on MIPS and includes a sophisticated SIMD instruction set extension to accelerate the Smith-Waterman algorithm. The algorithm is used in bioinformatics to align DNA, RNA and protein sequences. The generated processor is freely programmable and achieves a throughput of 2.6 billion cell updates per second (GCUPS) at an estimated power consumption of only 0.05 W.

Zusammenfassung

Viele digitale Systeme beinhalten Mikroprozessoren, weil sie aufgrund ihrer Programmierbarkeit sehr flexibel einsetzbar sind. Einige dieser Mikroprozessoren sind auf den jeweiligen Anwendungsbereich zugeschnitten, um die Ausführungsgeschwindigkeit von Programmen zu steigern und Energie zu sparen. Eine effiziente Entwicklung solcher anwendungsspezifischer Prozessoren bedarf geeigneter Entwicklungswerkzeuge.

Dazu trägt diese Arbeit, sowohl durch die Spezifikationssprache ViDL, als auch durch zwei Generatoren zur schnellen Entwicklung anwendungsspezifischer Prozessoren bei. Ein Entwickler spezifiziert einen Instruktionssatz in ViDL und generiert daraus einen Simulator, als auch eine mikroarchitektonische Implementierung eines Prozessors. ViDL verfügt über mächtige Konzepte, um eine Vielzahl an Instruktionssätzen auf einem hohen Abstraktionsniveau zu spezifizieren. So werden zum Beispiel die Pipeline und ihre Kontrolle nicht durch den Entwickler definiert, sondern automatisch durch den Generator aus dem Instruktionssatz hergeleitet. Um die praktische Anwendbarkeit von ViDL zu demonstrieren, wurden reale Instruktionssätze wie ARM, MIPS, Power und CoreVA spezifiziert und entsprechende Simulatoren und Prozessoren generiert. Eine ViDL Spezifikation ist der Notation in Instruktionssatz-Handbüchern sehr ähnlich und erlaubt deshalb eine schnelle Formalisierung (z.B. einen Tag für MIPS).

Der generierte Simulator führt durchschnittlich 60 Millionen Instruktionen pro Sekunde (Mips) aus, mit einer Spitzengeschwindigkeit von 140 Mips. Generierte Prozessoren erreichen eine Geschwindigkeit von ca. 600 MHz für eine 65 nm STMicroelectronics low power Technologie. Der Prozessorgenerator erzeugt eine n-stufige Pipeline, wobei n automatisch anhand einer vom Benutzer gegebenen Zielfrequenz ermittelt wird (z.B. 600MHz). Folglich können ohne zusätzlichen Aufwand verschiedenste mikroarchitektonische Implementierungen generiert werden. Alle Prozessoren und der Simulator sind garantiert konsistent, weil sie vollautomatisch aus derselben Spezifikation erzeugt werden. Ein ViDL Entwickler kann diese Konsistenz nicht verletzen, weder unbeabsichtigt noch vorsätzlich.

Die Eignung ViDLs zur Exploration von Entwurfsräumen und Erweiterung von Instruktionssätzen wurde durch die Entwicklung eines anwendungsspezifischen Prozessors im Rahmen dieser Arbeit nachgewiesen. Der Prozessor namens DNACore basiert auf MIPS und wurde durch einen Satz anspruchsvoller SIMD Instruktionen erweitert, die die Ausführung des Smith-Waterman Algorithmus erheblich beschleunigen. Der Algorithmus wird in der Bioinformatik eingesetzt,

um DNS-, RNS- und Proteinsequenzen zu analysieren. Der generierte Prozessor ist frei programmierbar und erreicht einen Durchsatz von 2.6 giga cell updates per second (GCUPS) bei einer Leistungsaufnahme von nur 0.05 W

Contents

Chapter 1

Introduction

1.1 Motivation

Digital systems are omnipresent in our today's life. They are for instance employed in personal computers, cell phones, cars, video game consoles and chip cards. Most of these digital systems include microprocessors, as they are flexible due to their programmability. The variety of different application areas raises the demand for customized processors. Processors in cell phones are supposed to consume low power, whereas desktop processors are optimized for high performance. A processor in a chip card includes special instructions to accelerate cryptographic algorithms, whereas processors in video game consoles use multimedia instructions for graphical operations. The application specific aspects of a processor are mainly encapsulated in its instruction set.

A multitude of such application specific processors is developed and increasingly employed in systems on a chip. Tools that support a rapid and reliable development of processors for a given instruction set are therefore essential. This

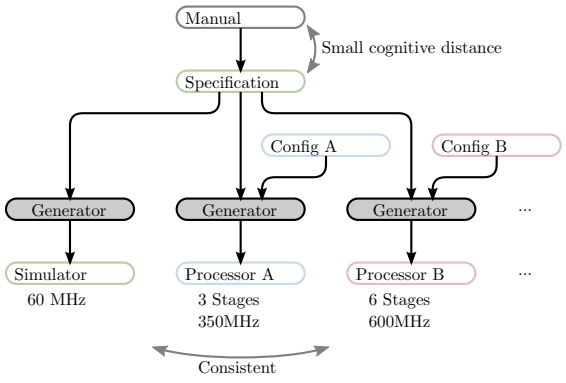


Figure 1.1: Overview of the proposed approach.

thesis proposes a system to generate a simulator and a processor implementation from a formal instruction set specification, as shown in Figure 1.1. The specification language enables rapid formalization of instruction set manuals, due to a small cognitive distance to descriptive manuals. The developer does not require particular knowledge of microarchitectures, since the specification language strictly abstracts from such aspects. The language excludes for instance all aspects of a pipeline and its control. This also eliminates a common source of faults. Thanks to this abstraction, processor implementations with different microarchitectures can be generated from the very same specification, as shown in Figure 1.1.

The generator automatically constructs an appropriate microarchitecture that suites the instruction set. The developer can control this derivation by an optional configuration. The configuration may for example constrain the clock frequency of the final processor. Based on such information and the specified instruction set, the generator determines for example the depth of the pipeline. Figure 1.1 shows the actual clock frequency and pipeline depth for two generated ARM processors.

As all generated implementations are based on the same instruction set specification, they are guaranteed to be compatible. Executing a program on each implementation will yields the same result. There is no way, a ViDL developer can break consistency of the generated implementations. The processor specification itself is always consistent and configurations have no effect on the processors execution semantics. As the generated implementations are consistent, extensive testing of processor implementations can be omitted.

Processor design flow In the first step of a classical design flow, the set of instructions is defined and evaluated using a simulator. The simulator basically emulates a real processor, but requires less development effort. It is typically programmed in C and runs on normal desktop computers. After the instruction set is fixed, a processor is defined in a hardware description language (HDL), such as VHDL. The description obeys a specific microarchitecture, which consists of structures and techniques to implement an instruction set. Note, that the same instruction set can be implemented with different microarchitectures. The resulting processors differ in power consumption, clock speed and chip costs, but can execute the same binary programs. Typically, a microarchitecture is selected that fits best the demands of the application area.

This design process has several disadvantages. The separate implementation of an instruction set in terms of a simulator and a hardware processor increases development costs and time to market. The instruction set or microarchitecture of the hardware implementation can hardly be changed in a late design stage. Decreasing for instance the length of the pipeline is laborious and error-prone. Functional units must be reassigned to stages, which affects pipeline registers and the pipeline bypass. As a result, the control unit must be adapted and even changes to branch prediction may be necessary.

In general, the development of pipelined implementations is complex due

to pipeline hazards. For this reason, processors need to be tested extensively. Development of test cases with a good coverage however is challenging. The microarchitecture must be considered to test instructions for all reachable pipeline states. In practice, testing has a major impact on development time and costs. Even the implementation of the simulator may be complex, due to the lack of bit-precise types and missing operations (e.g. "count leading zeros") in C.

Mixed level languages To overcome these difficulties, mixed-level processor description languages are proposed. Such languages specify a processor on the level of microarchitecture and the instruction set level. On the microarchitectural level, a rough layout of the processor is specified. This typically includes pipeline stages, functional units and register ports. The specification of an instruction then refers to these elements. The instruction set level therefore builds on the microarchitectural level. As a result, the whole specification is bound to one specific microarchitecture. Once the instruction set is specified, the microarchitecture can not be changed without major changes to the instruction set. For instance, changing a 5-stage pipeline to a 3-stage pipeline is expected to be very complex.

In mixed-level languages, pipeline control is typically specified explicitly. This allows for the description of complex and exceptional pipeline behavior. However, its specification is typically very complex and error-prone, as all pipeline states must be considered. Some exceptional sequences of instructions are easily overlooked, which may result in unintended bypassing or branch behavior. As the developer is responsible for the correctness of pipeline control, extensive testing is required. Some languages require in-depth knowledge on microarchitectures, like VHDL and C++. A developer must therefore have respective skills. Some languages impose an orthogonal structure on the instruction set. Instructions are then defined according to this structure. To formalize an existing instruction set manual, the developer has to analyze the instruction set to reconstruct regular structures. Some languages offer or even require a separate specification of instruction semantics for the simulator and hardware implementation. The developer is responsible to specify equivalent semantics for both. This may introduce inconsistencies between simulator and hardware. Hence, sufficient testing is required, to exclude such faults with a certain probability. Besides, the redundant specification reduces maintainability and increases development effort.

The outlined problems of classical processor design flows and mixed level languages are solved by ViDL and its generators. The language and the generators therefore implement a variety of new concepts and methods, which are summarized in the next section.

1.2 Overview

This thesis is structured as follows: The remainder of this chapter gives a basic introduction to the *context* and the *contribution* of this thesis. Its *fundamentals*

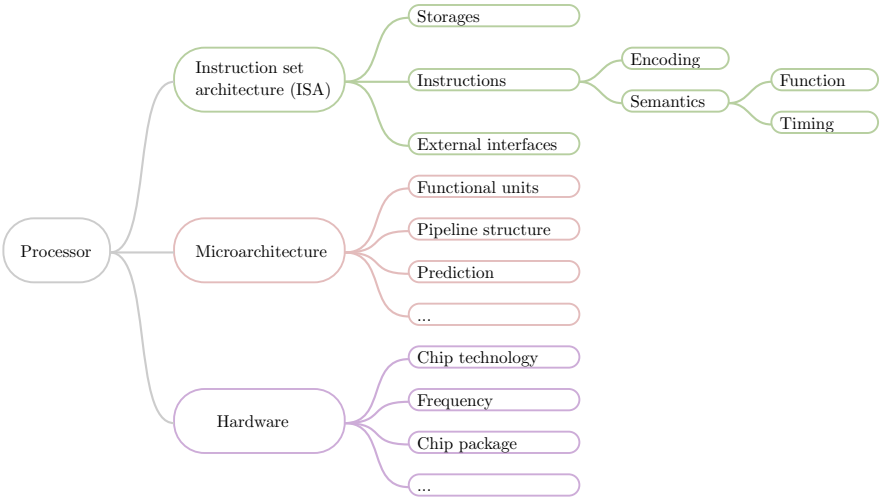


Figure 1.2: Structure of processor aspects.

by means of applied methods and concepts are briefly summarized, to introduce a common terminology and common definitions (Chapter 2). To give an overview of alternative approaches, *related specification languages* for instruction sets and processors are reviewed and classified (Chapter 3). Against this background, the *instruction set specification language ViDL* is proposed (Chapter 4). Explanations focus on ViDL's concepts and their impact on the quality of instruction set specifications. Practical applications of ViDL are demonstrated, using design patterns for frequent concepts of instruction sets as example (Chapter 6). Besides ViDL, the simulator generator and the processor generator account for a great part of this thesis. Major concepts and methods of both generators are presented (Chapter 7), including the timing driven generation of pipelined processors (Section 7.6). Both generators are evaluated, by examining the generated products for multiple real world instruction sets (Chapter 8). The speed of the simulator is measured and physical characteristics of the processors are estimated by synthesis tools. As an all embracing proof of concept, an application specific processor is developed for the Smith-Waterman algorithm, using ViDL and its generators (Section 8.6).

1.3 Processor aspects

This section describes the aspects of a hardware processor implementation. The purpose of this section is to introduce a uniform terminology and classification of processor aspects. A processor is defined by its instruction set, microarchitecture and hardware properties, as shown in Figure 1.2.

An *instruction set architecture* defines the aspects of the processor that are visible to an assembly programmer or compiler. Basically, it defines a set of

storages and instructions that operate on these storages. The set of storages typically includes a general purpose register file, a status register and a data memory. An instruction is defined by its encoding and its semantics. The semantics define the effect of executing the instruction on the processor state. They are typically denoted on a register transfer level. The terms "instruction set architecture", "instruction set" and ISA are used synonymously. Examples of popular instruction sets are Intel x86, Power, ARM and MIPS.

A *microarchitecture* comprises a set of techniques to implement an instruction set architecture. It has a major effect on the efficiency of a processor in terms of clock frequency, chip area and power consumption. A microarchitecture basically defines the structure of hardware units and their interaction. It defines for instance the structure of the pipeline and how branches are predicted. Hennessy and Patterson [24] use the term "Organization" as a synonym for "microarchitecture". As the latter appears to be more common, it is used in this thesis. An example of a microarchitecture is a 5-stage pipeline with a dynamic branch prediction using a 4096-entry 2-bit prediction buffer.

The last term "*hardware*" covers physical aspects of a hardware processor. This includes the used chip technology, clock frequency and chip package. The three classes of ISA aspects (ISA, microarchitecture and hardware) are independent. One instruction set can for instance be implemented using different microarchitectures. A low power microarchitecture may be used for mobile applications, whereas a deeply pipelined microarchitecture is used for high-performance applications. On the other hand, the same microarchitecture may be used to implement different ISAs. Processors that share the same ISA and microarchitecture may differ in their actual hardware implementation. For instance may the chip technology and the chip package be different.

1.4 Scientific contributions

This thesis contributes a series of concepts and methods, which are integrated in the processor specification language ViDL, the simulator generator and the processor generator. They are briefly introduced in the following, to give an overview of this thesis.

1.4.1 Language concepts

In contrast to existing approaches, ViDL strictly *abstracts from microarchitecture* (Section 4.3). Thanks to this abstraction, the microarchitectural design space can automatically be explored, driven by the processor generator (Section 8.1). Small processors, as well as deeply pipelined processors are generated from the same specification. Besides, abstraction enables generation of a consistent high-performance simulator (Section 8.4), since microarchitectural aspects need not be simulated.

To specify timing behavior of instructions, while abstracting from their microarchitectural implementation, the concept of so called *delays* is proposed

(Section 4.7). As a result, the number of delay slots of a branch can directly be denoted in ViDL, without considering the pipeline and its control. Views on storages and I/O ports can be defined using the concept of *architectural interfaces* (Section 4.8). It is a unified concept, which covers a variety of realistic register and storage structures, such as architectural registers and virtual address spaces (Chapter 6). The resulting abstraction improves clearness and maintainability of a specification. Architectural interfaces are based on equations and are translated into dataflow by the generator (Section 7.1.3). To efficiently define conditional and partial write accesses to storages, the unified concept of *epsilon logic* is proposed (Section 4.6). Epsilon logic is an instance of multi value logic, using three states, namely "zero", "one" and "epsilon", which means "don't modify". The concept integrates well in functional languages (such as ViDL) and improves clearness and reliability of specifications. Besides, a method is proposed, which translates epsilon logic into binary logic, as needed for implementations (Section 7.4.2).

As bit-widths need not be defined in ViDL, a type system and a respective type-analysis are proposed, to derive the widths of operations (Section 4.9). The type system is based on polymorphism and subtyping. In contrast to Verilog, bit-width rules on operations can concisely be specified in terms of signatures. Type analysis is based on type inference techniques. It statically checks for soundness and detects ambiguous semantics, as contained in the pseudo-code of the ARM manual (Section 4.9.3). Due to implicit typing, instruction set specifications are simplified and maintainability is increased. For instance, the widths of an entire instruction set can be changed, by simply redefining the width of registers (Section 8.2.4). ViDL integrates concepts of functional languages, such as tuples and higher order functions (Section 4.5). The language is free of side effects, which adds to reliability and clearness. Functionals can be defined, to encapsulate concepts of instruction sets, such as SIMD (Section 6.10).

1.4.2 Generation methods

This thesis contributes methods to *generate pipelines* from specifications of instruction sets (Section 7.6). In contrast to related work, the approach is fully automated, i.e. no microarchitectural aspect of the pipeline needs to be specified by developers. Instead, the entire microarchitecture is contributed by the processor generator, based on a user-defined timing constraint (Section 7.6.3). The methods construct the entire pipeline, including forwarding circuits, interlocking, as well as control for branch instructions. All hazards are properly resolved in the resulting pipeline, while minimizing instruction latencies. The methods are based on scheduling techniques, term rewriting, as well as partial functions and their analysis. They are sensitive to instruction semantics and to the propagation delay of the targeted chip technology.

As ViDL abstracts from register ports, a *port allocation* method (Section 7.6.1) is proposed and integrated in the processor generator. The method assigns read and write accesses of instruction semantics to read and write ports of register files and memories. The number of ports is minimized, while excluding resource

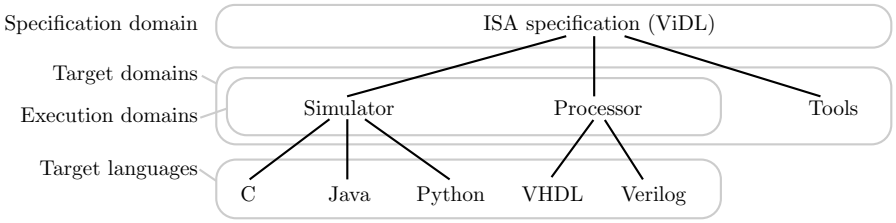


Figure 1.3: Domains and their relation.

conflicts. Besides, concurrent accesses are merged, if they refer to the same address. The allocation problem is reduced to graph coloring.

The thesis shows, how instruction semantics are systematically transformed and optimized using *term rewriting* on *dataflow graphs* (Section 7.3). A comprehensive set of rewrite rules has been defined, which encapsulates expert knowledge on semantic equivalences. The set is extensible and clear, which greatly adds to reliability of the generator. Partial evaluation of expressions is for instance solved using term rewriting (Section 7.4.1).

Code of the simulator is generated such, that it utilizes the principle of *lazy evaluation* (Section 7.5.4). Only those parts of instruction semantics are simulated that contribute to the result of an instruction. For instance, the target address of a branch is only computed, if the branch is actually taken.

The thesis presents methods to process *arbitrary and ultra-wide* instruction semantics (e.g. 257 bit). All parts of the generation process are considered, including static evaluation and code generation for C. Static evaluation is implemented in the generator using an arbitrary precision library (Section 7.3.4). The generation of C-code is based on operation specific generator algorithms. The algorithms break ViDL operations of arbitrary width down to 32-bit or 64-bit integer arithmetic in C (Section 7.5.2).

1.5 Processor implementations

In my approach, an ISA is defined in the *specification domain*, using the language ViDL. From this specification, products of different *target domains* are generated, as illustrated in Figure 1.3. In particular, this is the *simulator* domain, the (hardware) *processor* domain and the *tools* domain.

The simulator and processor domain are called *execution domains* in the following, as their products can actually execute ISA compliant programs. The simulator and processor domain are separated, as they significantly differ in structure. A simulator is described in terms of data structures and sequential algorithms. A processor in contrast is defined by inherently parallel units and their interconnection. The gap between both domains is further increased by complex processor microarchitectures, which are omitted in the simulator for the sake of efficiency.

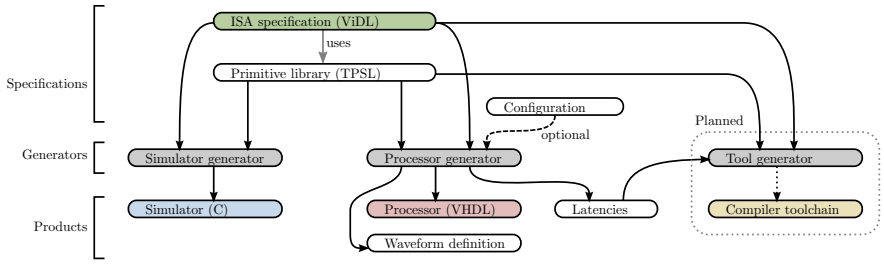


Figure 1.4: System of specifications, generators and products.

The tools domain covers ISA specific tools, which are used for software development. A classical toolchain for an ISA consists of a C-compiler, an assembler, a linker and a disassembler. This domain significantly differs from the former ones, as tools only consider certain properties of the ISA. An assembler for instance considers the encoding of instructions, but not their behavior.

The products of the execution domains can be implemented in different languages. A simulator is typically implemented in imperative programming languages, such as C, Java or Python. The structure of a processor is defined using hardware description languages, like VHDL or Verilog. For the tools domain, a target language corresponds to a specific retargetable toolchain. The generator may produce ISA descriptions that are specific for different toolchains.

Currently, generators are available for both execution domains and the most typical target languages C and VHDL. As ViDL is not bound in any way to these languages, generators for further target languages can be added. The effort for retargeting can be considered low, as only the code generation needs to be adapted. Domain specific concepts, such as pipelining or lazy evaluation, remain unchanged. The development of a generator for the third domain "tools" is not part of this thesis. It is discussed as future work in Chapter 9.

1.6 System overview

This section is intended to give a brief overview of the entire system of specifications, generators and their products on a high level. The in-depth discussion of its components is postponed to subsequent sections. The basic idea of the system is to generate different processor implementations from a central ISA specification, as illustrated in Figure 1.4.

An ISA is specified using the *Versatile ISA Description Language* (ViDL). The specification defines storages, instructions and external interfaces of the ISA. An important feature of ViDL is that *only* aspects of the ISA are specified. Microarchitectural aspects, such as pipeline stages and register ports are not defined. ViDL is described in detail in Chapter 4. For evaluation, the instruction sets of ARM, MIPS, Power, CoreVA, and SRC have been specified (Section 8.2.1). In addition, a one instruction set computer (OISC) and an

instruction set extension (ISE) for DNA sequence alignment have been defined.

ViDL uses primitives such as addition (`add`) or sign extension (`extz`), to define the semantics of instructions. The signatures and semantics of primitives are defined in the separate *primitive library*, which is specified using the *transfer primitive specification language* (TPSL, Chapter 5). It should be noted, that most ISA developers do not have to consider TPSL, as they can simply use the existing primitive library, which includes about 50 primitives. In rare cases, the developer may extend the library by additional special purpose primitives.

The ISA specification and primitive library are fed into generators. The *simulator generator* (Section 7.5) produces an efficient *instruction set simulators* (ISS) from this input. Simulation speeds of 60 Mips are likely on a Pentium 3 GHz workstation (Section 8.4).

The *processor generator* (Section 7.6) produces a *processor*, i.e. a microarchitectural implementation of the ISA. The microarchitecture of the processor can be controlled using an optional *configuration*. The configuration typically defines the targeted clock frequency of the processor. The generator then estimates propagation delays and selects an appropriate pipeline depth to meet the targeted timing. The configuration enables the developer to generate multiple processors with different microarchitectures from the very same ISA specification. All processors and the simulator are guaranteed to be consistent. They can execute the same binary programs and yield the same results. The number of required execution cycles however may differ due to different dynamic behavior. The dynamic behavior is mainly affected by instruction *latencies*. Information on latencies is emitted by the processor generator and may be utilized by the scheduler of a generated processor to avoid interlocking. Another product of the processor generator is a waveform definition file for ModelSim. It contains the identifiers and structure of significant signals, which are then visualized in ModelSim.

1.7 Evolution of ViDL and its generators

The system described in the last section is the result of an extensive development process, which is illustrated in Figure 1.5. In particular ViDL is the outcome of a long process of ISA exploration and evaluation. It is based on the specification language UPSLA, from which it borrows the concept of encoding patterns and distinct specification of instructions.

To derive the underlying instruction set model of ViDL, a large set of representative instruction sets has been studied. Common concepts have been identified and generalized. High level concepts in instruction sets have been broken down to a combination of basic concepts. Conditional execution is for instance modeled as a combination of conditional expressions and epsilon logic. The development of the ISA model was probably one of the most complex tasks.

With the language in mind, representative sets of ISAs have been specified. This way, missing features and inadequate concepts have been identified in an early stage of development. For instance, the need to make the set of primitives

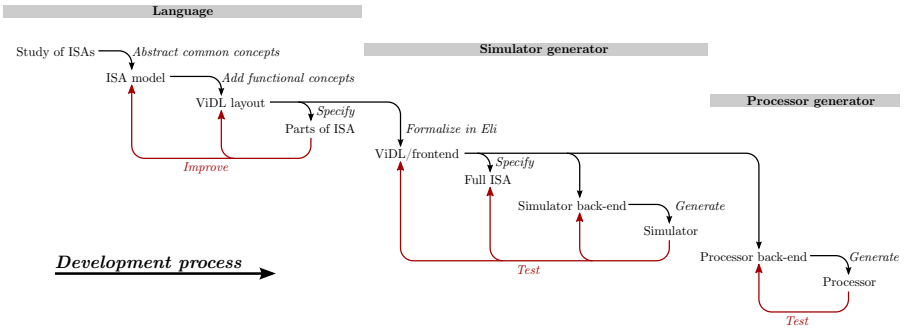


Figure 1.5: Development process of ViDL and its generators.

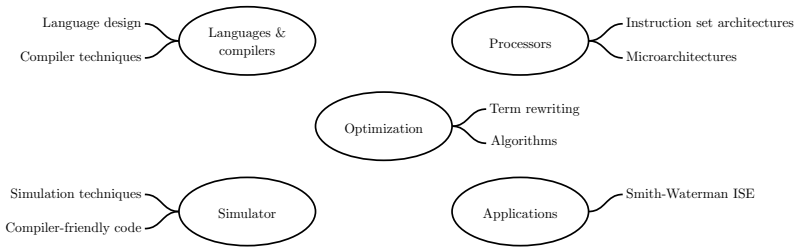


Figure 1.6: Involved problem areas.

extensible has been detected during this process.

After the model and the respective language had annealed to a sufficient degree, the generator has been developed. At first, the front-end and the simulator back-end have been developed, for three reasons.

- The simulator offers a convenient way to test the front-end and common optimizations of the generator.
- Existing experience in the area of simulator generation from UPSLA and the CoreVA architecture.
- The structure of the simulator is much simpler, as it does not involve the generation of a pipelined microarchitecture.

After a correct baseline simulator had been generated from an ARM specification, the processor back-end has been developed. Finally, the generated products were repeatedly evaluated and optimized.

1.8 Areas of expertise

The development of the generator system is an inter-disciplinary task. It requires expertise in very different problem areas, as illustrated in Figure 1.6.

The first major problem area covers processors and microarchitectures. A representative set of ISAs and their common structures and concepts has to be understood, to derive a simple, yet powerful ISA model. This domain specific knowledge is crucial for the design of ViDL. For code generation, expertise on state-of-the-art microarchitectures is required. Concepts such as pipelining, forwarding, interlocking and branch prediction must be implemented as generalized algorithms in the generator. This requires an in-depth understanding of these concepts, as correctness must be considered for any instruction set and not only for a specific one.

Language design mainly requires knowledge on language concepts and their benefits. A good selection of language concepts is a non-trivial task, as language qualities such as simplicity, reuse, maintainability and readability must be considered. In particular, concepts of functional languages and their implementation have to be understood, as ViDL inherits a large subset of SML. The language has to be formal and unambiguous, to be suited for generation. This implies a basic understanding of compilers and the target domains. To implement the generators, compiler development skills are required. This includes context free grammars, name analysis, type analysis and code generation. The grammar needs to obey the parser generator's grammar class, which is LALR(1) for ViDL. Name analysis in ViDL is a non-trivial task, as highly recursive scoping rules are inherited from SML. Type analysis requires expertise on type inference, polymorphism and subtyping. The type system also involves *formal proofs*, to ensure soundness. In general, the demand for correctness and generality makes compiler development hard.

To yield efficient products, the ViDL generators include sophisticated optimizations. The implementation of these optimizations involves the development of non-trivial algorithms and data structures. The pipeline scheduler 7.6.4 and the port allocator 7.6.1 are examples of such algorithms. The integrated term rewriting system requires knowledge on equivalent register transfer, and confluence.

To produce an efficient simulator, appropriate simulation techniques have to be applied. For example, the simulation speed is significantly increased by a decoder cache and a lazy evaluation strategy of instruction semantics. Besides, technical aspects of the generated C-code play an important role for efficiency. A basic understanding of C-compilers and their optimizations is required, to produce compiler-friendly C-code. For instance, the use of separate compilation units typically result in conservative estimates on data-dependencies, which finally limit the scheduler.

To evaluate ViDL's fitness for instruction set extension, the MIPS instruction set was extended by application specific instructions for the Smith-Waterman algorithm. To develop this extension, the structure of that algorithm had to be understood in depth. Besides, a basic idea of its application for sequence alignment of DNA, RNA and proteins is required.

In addition to expertise on concepts, methods and algorithms, an extensive technical knowledge is required to implement the generator system. For instance, 13 different languages were involved in the development of the whole

system. This includes declarative languages (ELI, Make and flex/bison), imperative languages (C and C++), scripting languages (Perl, Python and Bash), assembly languages (ARM, MIPS and Power), the functional language SML and the hardware description language VHDL.

Chapter 2

Fundamentals

This section outlines the fundamentals of this thesis and introduces a common terminology. The development of the language ViDL and its generators is founded on knowledge and techniques of several scientific disciplines. For instance, features of characteristic instruction sets (Section 2.1) and typical processor design scenarios (Section 2.2) have significantly guided the development of ViDL. Besides, ViDL obeys design guidelines of domain specific language (Section 2.3). The ViDL compilers implement classical compiler techniques (Section 2.4), sophisticated analyses (Section 2.5) and transformations (Section 2.6) to optimize the generated product. To derive a pipelined processor implementation, structures and principles of modern microarchitectures (Section 2.7) are utilized.

2.1 Instruction set architectures

To understand the design of ViDL, it is important to have a basic understanding of realistic instruction sets and their characteristics. This section therefore introduces two major instruction sets, namely ARM and MIPS, which have (among others) significantly influenced the development of ViDL. The instruction sets have inspired several concepts of ViDL. The goal has been to design the language such, that both real world instruction sets can precisely be specified, including irregularities and uncommon aspects. The evaluation in Chapter 8 presents the results for the formalization of both instruction sets.

In general, an instruction set architecture defines a set of instructions and storages. It thereby determines the execution semantics of binary programs. The description of an instruction set is for instance used by assembler programmers to write assembly programs. An instruction set can be regarded as an interface for the execution of binary programs. Processors that implement the same instruction are compatible. Executing a program on these processors will therefore yield the same results. However, the execution speed and the microarchitectural implementation of the processors may differ.

The purpose of this section is to give an impression of exceptional instruction set characteristics. In particular, the ARM, MIPS and an OISC instruction set is regarded. This section is not intended to discuss these instruction sets in detail, but only their interesting aspects. The presented aspects are considered to be complex in terms of instruction set specification. A specification language should be powerful enough to define these characteristics precisely and concisely.

2.1.1 ARM

The ARM instruction set [2] belongs to the class of RISC instruction sets. It has been chosen for evaluation, as it is a popular and state-of-the-art instruction set for mobile applications, which became more important in recent years. The instruction set is basically quite regular, but also includes some uncommon characteristics, which challenge ViDL and its generators. In the following, some of these irregularities of ARM are briefly described.

2.1.1.1 Conditional execution

All instructions can be *executed conditionally*. Along with each instruction, a condition mode can be specified. The condition mode corresponds to a predicate on the status flags (Z,N,C,V) of the processor. If the predicate holds, the instruction is executed. Conditional execution is used to eliminate control flow in a program. The basic idea is that conditional execution of few instructions is more efficient than a branch, due to control-hazards. This is especially true for deeply pipelined microarchitectures and simple branch prediction.

2.1.1.2 Shifter operand

Most ARM instructions have two operands. The second operand is the so called *shifter operand*. Each arm instruction can apply a shift operation on this operand. For instance, an add instruction on two operands `x`, `y` actually performs the computation `x+shiftOperation(y)`. The operand `y` is either an immediate or a register value. The definition of the shift operation is surprisingly complex. It uses one of 5 shift modes, such as “logical shift left” or “arithmetic shift right”. The set of modes even includes an extended 33-bit rotate operation, which involves the carry flag. The shift distance is given by an even immediate value.

2.1.1.3 Registers

The ARM instruction set defines a set of 16 general purpose registers, which are accessed by most instructions. The last register (`r15`) is defined to be the program counter. Write accesses to this register therefore result in branches. As a result, any instruction that writes to `r15` is actually a branch instruction. Some of the other general purpose registers are dynamically reconfigurable. Depending on the processor mode (e.g. user-mode, supervisor-mode, interrupt-mode), these general purpose registers are associated with different physical storage.

2.1.1.4 Exceptional instructions

The ARM instruction set defines usual 32-bit arithmetic and logical instructions. Besides, it defines some exceptional instructions, such as 64-bit multiplications, multi-cycle load/store-multiple instructions and atomic swap instructions.

2.1.2 MIPS

The MIPS instruction set [28] is a rather simple and regular RISC instruction set. It uses only few exceptional concepts, which are outlined in the following. The specification of MIPS has primarily served as a proof of concept, to show that regular instruction sets can efficiently be specified in ViDL. Actually, the MIPS instruction set has been specified in only one day, as described in Section 8.2.2.

2.1.2.1 Multiplications

Most instructions read parameters from general purpose registers and write their result to general purpose registers. However, 64-bit multiplications use a dedicated 64-bit wide special purpose register. This register can not be accesses by "normal" instructions. The instruction set therefore includes dedicated instructions to transfer values between the 64-bit register and the general purpose registers.

2.1.2.2 Branches

All branch instructions have one delay slot, i.e. the instruction following a branch is always executed. This behavior must precisely be modeled by an instruction set specification, as it affects program semantics. The set of branch instructions includes less common *section-relative* branches. The target of these branches is the sum of the branch's section address and an offset.

2.1.2.3 Zero register

The first general purpose register (`r[0]`) is always read as zero. The register can be used as any other register, but must not be written. As the state of the register is never changed, it does not have to be implemented by a hardware register. An instruction set specification language should provide concepts to define constant registers. Otherwise, the generated hardware may include unnecessary registers.

Besides, typical RISC instructions, the MIPS instruction set defines instructions to operate on bit-slices. These instructions are used to access an arbitrary sequence of bits in a register. The range of the sequence is given by immediates and is therefore not known statically.

2.1.3 OISC — One instruction set computer

The instruction set of a "one instruction set computer" consists of exactly one instruction. An OISC has been specified in ViDL to demonstrate, that small instruction set specifications lead to small generated processors. This section briefly outlines the idea of OISCs and discusses some of their characteristics.

Although an OISC consists of only one instruction, it is Turing complete which means that any program can be expressed in terms of an OISC instruction set. An OISC is exceptional in two ways: First, its definition is very short and second, the defined instruction combines behavior of different instruction classes, such as load, arithmetic and branch.

Several flavors of OISCs have been proposed [18], which differ in the semantics of their instruction. The most common OISC defines a "subtract and branch if negative" (`SBN`) instruction. It was first proposed by van der Poel [52]. The SBN instruction has three parameters A, B and C and is defined by the following imperative code.

```
SBN(A,B,C)
{
  MEM[A] := MEM[A] - MEM[B]
  if MEM[A] < 0 then PC := C
}
```

The instruction subtracts two values from memory and writes the difference back to the minuend. If the difference is negative, the instruction branches to an absolute target address. In terms of classical RISC instruction sets, the instruction combines a load instruction, a store instruction, a subtraction and an absolute conditional branch. It can directly be defined in ViDL to generate a respective simulator.

However, it can not efficiently be implemented as processor for two reasons. First, it does not utilize registers, which allow for fast accesses. Second, the instruction includes data dependent read and write accesses to the main memory. It can therefore not be implemented using synchronous memory.

To circumvent these restrictions, this thesis uses a modified OISC instruction set. It defines one instruction memory and one register file with two read ports and one write port. The SBN instruction only operates on the register file. Data and instructions are thereby separated, similar to a Harvard-architecture. To initialize the register file, a "load immediate" instruction is added. Alternatively, one may define the registers to contain the constant $c = 1$ or $c = -1$ on initialization.

2.2 Design scenarios

During this thesis, several different processor and ISA design scenarios have been identified. These use cases have been considered during the development of ViDL. A design goal of ViDL is to support the developer in these scenarios.

This section is therefore essential to understand the design of ViDL. It is also a foundation for the review of related specification languages in Chapter 3.

Instruction set extensions (ISE)

Some classes of algorithms include specific computations, which can not efficiently be mapped to the usual set of general purpose instructions. To accelerate these algorithms, an existing instruction set is enriched by application specific instructions. The instructions should be generic to some extend, to accelerate not only one specific algorithm, but a class of applications. At its best, this also includes future algorithms of the specific area of application.

In the ISE scenario, an existing ISA is extended by instructions and may also be extended by special purpose registers. The resulting ISA has to be backward compatible, such that legacy software can be executed and existing compiler tool-chains can be used. To enable backward compatibility, additional instructions are embedded into unused regions of the instruction space. The behavior of existing instructions remains unchanged. To demonstrate, that ViDL is suited for this scenario, an existing MIPS specification has been extended by instructions and special purpose registers, yielding the DNACore instruction set (Section 8.6).

Design space exploration (DSE)

A DSE is performed to tailor a processor to a specific area of application. Therefore, certain dimensions of the ISA as well as the microarchitecture are explored. Typical dimensions of the ISA include the number of registers, the size of storages, the width of the datapath, the set of instructions and the delay of instructions. Interesting dimensions of the microarchitecture include the pipeline depth, forwarding, branch prediction and the implementation of functional units (e.g. multi-cycle multiplier vs. pipelined multiplier). As part of evaluation, the ISA design space (Section 8.2.4) and the microarchitectural design space (Section 8.5.3) have been explored.

Implementation of legacy ISAs

There are legacy processors, which are still in use, but no longer supported and commercially available (e.g. Zilog Z8000 [57]). The implementation is defined in an out-of-date language and its microarchitecture is superseded by modern approaches. In this design scenario, a compatible processor is developed using a state-of-the-art microarchitecture. The foundation of development is the existing ISA manual.

ISA design from scratch

The ISA is developed from scratch, based on the intended area of application. The designer has a rough idea of the new ISA and the applied concepts. All design decisions are freely made without considering existing designs.

Improvement of microarchitecture

To improve the efficiency of a processor implementation, it is desirable to incorporate novel approaches in microarchitecture design into existing processors. This scenario may range from a minor local modification, which can easily be integrated into an existing implementation, to a major change, which necessitates a complete reimplementation of the processor. The resulting implementation must be compatible with respect to the ISA.

2.3 Domain specific languages

This section gives a brief introduction to language design guidelines for domain specific languages. They are used in Chapter 3 to review related specification languages. Besides, they have significantly influenced the design of ViDL. The guidelines are based on the books by Watt [54] and Sebesta [45]. Watt and Sebesta mainly focus on programming languages, but their language design guidelines can also be applied to ISA specification languages to some extend. In the following, language aspects that are essential for ISA specification are briefly summarized.

Simplicity

A language should be simple from the user's point of view. This should also hold for new users. A steep learning curve may discourage developers that are new to the language. The language should be readable, i.e. the meaning of a specification should immediately be clear. The language should also be writable, which means that it should be easy to express a thought in terms of the language. The semantic gap between the developer's imagination and the language should be small. A language should provide a small set of generic and orthogonal constructs, instead of an extensive set of specialized and similar constructs. The constructs should be regular, which means that they can be combined arbitrarily to specify complex aspects of an instruction set. Chapter 4 discusses the concepts of ViDL with respect to simplicity.

Reusability

Reuse is a major objective of engineering. The reuse of existing and tested components saves time, avoids flaws, and increases maintainability. Typically, reuse involves some kind of abstraction.

In software engineering for instance, functions are used to abstract from expressions. A function can then be called in different contexts with different arguments. Further concepts for reuse in software engineering are procedures, ADTs, classes, libraries and frameworks. In hardware engineering, similar concepts have emerged, such as entities, libraries (e.g. Design Ware Components) and IP cores (e.g. complete processor cores). The concepts for reuse in ViDL are primarily discussed in Section 4.5.

Portability

To enable portability, a language should abstract from the target domain. For instance, the programming language C abstracts from the instruction set of a processor. A standard C program can therefore be compiled for different targets, such as x86 or PowerPC. The abstraction is provided by the compiler, which translates a C program into processor dependent machine code. The C program is therefore portable. However, the C language also demonstrates how easily portability can be broken. Most C Compilers feature a way to bypass abstraction by means of inline assembly, which belongs to the target domain. Abstraction and portability are thereby broken, which means that the code is not platform independent.

To guarantee portability, an ISA specification language should strictly abstract from target specific aspects. For instance, a processor specification should not define microarchitectural aspects, to allow for generation of different microarchitectural implementations. Besides, the specification should not contain fragments of C or VHDL code that are literally copied into the generated code. This would prevent generation of implementations in different target languages, such as VHDL or Verilog. Section 4.3 describes how ViDL realizes a thoroughgoing abstraction, which guarantees portability.

Completeness

Typical general purpose programming languages are Turing complete, which means that any program that can be expressed in programming language X can also be defined in programming language Y. Completeness needs typically not be considered for general purpose programming languages and is not covered by Watt and Sebesta.

Domain specific languages however use a higher level of abstraction and thereby typically restrict the area of application. An aspect that can be expressed in a DSL X may not be expressible in a DSL Y and vice versa. Of course, bypassing techniques may be used as a back door to widen the applicability of the DSL. However, bypassing breaks abstraction and portability, as described earlier.

For an ISA specification language, the completeness is determined by the underlying ISA model. A very general model covers a large set of ISAs, but may restrict portability and efficiency. The ISA model of ViDL is discussed in Section 4.2.

Reliability

Reliability means, that errors and flaws are avoided by the design of the language or detected by analyses of a compiler (respectively generator). The generator should check the input for errors, including static semantics. An undefined aspect should be detected and reported along with a meaningful error message to allow for rapid debugging.

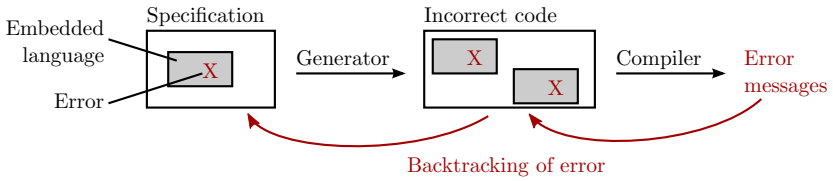


Figure 2.1: Tedious backtracking of errors as a result of embedded languages.

The ViDL generator for instance uses a powerful type-system to check the soundness of bit-widths (Section 4.9) as part of static semantics. In contrast, the embedding of target language code fragments makes such analyses virtually impossible. Figure 2.1 shows an example, where the code of an embedded language contains an error X. The embedded language is not under the generator's control and therefore not checked. The generated code includes a copy of the embedded code, which is incorrect. Hence, the generated code is refused by subsequent tools and tedious backtracking is required to relate the reported error to its source.

Consistency

In general, a source language may be translated into different implementations and different target languages. A C program may for instance be compiled for ARM and for Intel processors. To increase performance, the program may contain target specific blocks of inline assembly, one for ARM and one for Intel. These blocks are independent but are supposed to implement the same behavior. This bypassing of target specific code has two major disadvantages. First, the development effort is increased, as the same behavior is specified multiple times in different languages. Second, the developer is responsible to write semantically equivalent code.

In the context of an ISA specification, redundant C and VHDL code may lead to different behavior of the simulator and the microarchitectural processor implementations. Hence, excessive testing of the implementations is required, as described in [13]. Since ViDL does not require alternative specifications for different implementation domains, consistency is guaranteed.

2.4 Compilation methods

As part of this thesis, two generators have been developed — a simulator generator and a processor generator. The simulator generator translates a ViDL specification into a processor simulator, which is implemented in the imperative programming language C. The processor generator translates a ViDL specification into a hardware description of a pipelined processor. The structures of the target codes are very different from the source language. The generator therefore requires sophisticated compiler techniques for translation and optimization.

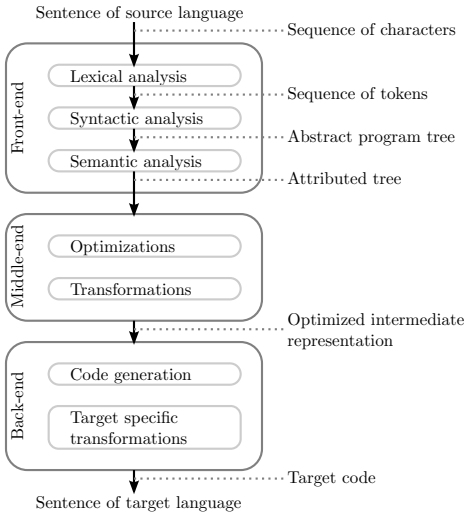


Figure 2.2: Generic structure of a compiler.

This section is intended to give a basic overview of these techniques. In-depth information can be found in literature [30, 32]

In general, a compiler translates a sentence of a *source language* into a semantically equivalent sentence of a *target language*. A C-compiler for instance may translate a sentence of the source language "C" into an equivalent sentence of the target language "Intel x86 Assembler", where the sentence is a program. In the following it is assumed, that the source language is a programming language. However, the same applies to other formal languages.

The generic structure of a compiler is shown in Figure 2.2. It consists of a front-end, a middle-end and a back-end. The *front-end analyzes* the sentence of the source language and yields an *intermediate representation (IR)* of the sentence. The *middle-end* optimizes the representations using sophisticated analyses and transformations. The *back-end* finally produces the target code. This step is also known as *code generation*. Before generating code, the back-end may also apply target specific optimizations on the intermediate representation.

2.4.1 Front-end

The front-end of a compiler can be further subdivided into the lexical, syntactic and semantic analysis. During *lexical analysis*, a *scanner* transforms the sentence of the source language (sequence of characters) into a sequence of *tokens*. A token represents an atomic element of the source language, such as a keyword, identifier or literal. The C-code "`return a+1`" is for instance divided into a sequence of 4 tokens. The tokens `return` and `+` are keywords, `a` is an identifier and `1` is a literal.

During *syntactic analysis*, a *parser* structures the sequence of tokens. The

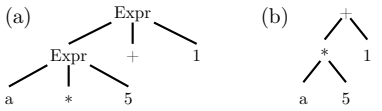


Figure 2.3: Example of (a) an abstract syntax tree and (b) a corresponding Kantorovich tree (b).

result is an *abstract syntax tree (AST)*, which reflects the structure of the sentence. For example, the abstract syntax tree of the C-expression `a*5+1` is shown in Figure 2.3a and is similar to its Kantorovich tree in 2.3b.

During *semantic analysis*, static properties of the abstract syntax tree are analyzed. The analysis uses a *tree walking algorithm* to annotate nodes of the AST with attributes. Semantic analysis typically includes *name analysis* and *type analysis*. Name analysis *binds* identifiers to so called *entities*. The uses of an identifier "`a`" are for instance bound to the respective variable. Note, that in most languages, different entities may share the same identifier. Name analysis therefore needs to consider the language's *scope rules*.

The second major semantic analysis is that of types. For a *statically typed* language, type analysis determines the types of program entities, such as variables and functions. For simple *explicitly typed* languages, types may be determined by a tree walking algorithm. For *implicitly typed* languages like SML, types are determined by *type inference*. Basically, type inference solves a system of equations, where a variable represents the type of a program entity. Equations express type constraints between these entities. A solution of the system of equations is thereby a valid assignment of types to entities. Type analysis is explained in Section 2.5.

2.4.2 Middle-end

The middle-end implements *optimizations* on the intermediate representation. At its best, these optimizations are independent from source and target languages. An optimization transforms the intermediate representation such, that certain characteristics of the resulting target sentence are improved. For instance, a C-compiler includes optimizations to increase the execution speed of the resulting program. A valid optimization must preserve the semantics of the intermediate representation. Its interpretation before and after optimization must be the same. Examples of optimizations include *dead code* elimination, *evaluation of constant expressions* and *function inlining*.

2.4.3 Back-end

Code generation in the back-end translates the intermediate representation into code of the target language. In case of source to source translation, a *pattern based* code generator may be used. Such a generator uses text patterns of the

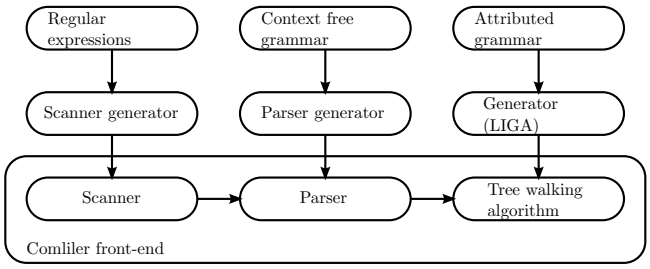


Figure 2.4: Generation of compiler components.

target language, which include insertion points for other fragments of target code.

2.4.4 Compiler framework

Compiler frameworks enable the efficient implementation of compilers [32]. Significant parts of the front-end can be generated, including the scanner, parser and tree walking algorithm. Figure 2.4 shows an overview of the front-end, its components, their specification and the involved generators. The *Eli compiler framework* [31] supports all three components shown in the figure. The generators are well integrated, i.e. a developer does not have to care about the interface between compiler stages. In contrast, the combination of *Flex and Bison* [9, 42] covers only the lexical and syntactic analysis. An advantage of Flex is, that different scanner modes can be defined, where each mode activates a certain set of tokens. During analysis, the mode can be changed to activate different subsets of tokens. This feature is required for embedding of languages, such as embedding C in a domain specific language.

2.5 Type systems

In ViDL, each value is a bit-string. The widths of such bit-strings needs to be known for code generation. However, it is not specified explicitly by the developer in ViDL. Instead, bit-widths are determined by the generator. Therefore, they are modeled by a type system and inferred by type-inference.

This section gives the foundation for ViDL's type system, which is discussed in Section 4.9. In computer science, a *type* is defined by a *set of values* and *operations* on these values [45]. A 32-bit unsigned integer type represents for instance the set of integral numbers from 0 through $2^{32} - 1$

$$\text{setOf}(\texttt{uint32}) = \{x | x \in \mathbb{Z} \wedge 0 \leq x < 2^{32}\}$$

Operations on this type include 32-bit addition, and multiplication.

2.5.1 Subtyping

If any value of a type S is also a value of a type T, the type S is said to be a *subtype* of T, denoted $S <: T$. The subtype relation on types is equivalent to the subset relation on the respective set of values.

$$S <: T \Leftrightarrow \text{setOf}(S) \subseteq \text{setOf}(T)$$

As a result of this relation, a value of type S may be given, where a value of type T is expected. The 16-bit unsigned integer type is for instance a subtype of the 32-bit unsigned integer type.

$$\texttt{uint16} <: \texttt{uint32}$$

2.5.2 Tuples

A sequence of n values[1] is called a *tuple* of arity n, or just n-tuple. The 3-tuple $(1, 2, 3)$ consists for example of the *components* 1, 2 and 3. Note, that the order of components is significant, i.e. $(1, 2)$ is different from $(2, 1)$. The type of an n-tuple is given by the cross product of the component's types. A pair of a 16-bit integer and a 32-bit integer has for example the type $\texttt{uint16} \times \texttt{uint32}$. Two tuple types are in subtype relation, if both types have the same arity and if the respective component types are in subtype relation.

$$S_1 \times \ldots \times S_n <: T_1 \times \ldots \times T_m \Leftrightarrow n = m \wedge \forall i : S_i <: T_i$$

Tuples are used to model functions with multiple parameters and results. A function with three parameters and two results is regarded as a function that expects a 3-tuple and yields a 2-tuple. According to this model, a function always has exactly one parameter and one result, where the parameter or result is a tuple. This model is used in the following and for the discussion of ViDL's type system.

2.5.3 Signatures

The type of a function is called *signature*. The signature defines the function's *parameter type* and *result type*. A 32-bit add-operation has for instance the following signature.

$$\texttt{add32} : \texttt{uint32} \times \texttt{uint32} \mapsto \texttt{uint32}$$

The function expects a pair (2-tuple) of 32-bit integer values and yields a 32-bit integer value (1-tuple).

[1] n is static

2.5.4 Polymorphic types

A *polymorphic type* includes one or more *type variables*. Binding values to these variables yields a concrete type. An n-bit unsigned integer type is an example of a polymorphic type, where n is the type variable. The set of values of this type depends on the actual value of n

$$\text{setOf}(\texttt{uint<n>}) = \{x | x \in \mathbb{Z} \wedge 0 \leq x < 2^n\}$$

A polymorphic type represents a set of types, one type for each instantiation of type variables.

2.5.5 Polymorphic functions

If a signature contains type variables, the respective function is said to be *polymorphic*. An n-bit add operation for instance is polymorphic, as its signature uses the type variable n.

$$\texttt{add<n>} : \texttt{uint<n>} \times \texttt{uint<n>} \mapsto \texttt{uint<n>}$$

2.6 Term rewriting systems

A term rewriting system (TRS) transforms terms based on a set of so called rewrite rules. Transformations typically simplify a term according to some cost estimate. The ViDL generators use a TRS for this purpose, i.e. to simplify and transform the intermediate representation of an instruction set (see Section 7.3). The intermediate representation of the instruction set is therefore regarded as a term. A set of rewrite rules is defined to simplify and normalize this representation. The rules express equivalences on bit-string operations, i.e. the semantics of the intermediate representation are not affected by transformations. In the following, a brief overview of term rewriting is given. This section is only meant to give an overview and introduce a common terminology. Literature on term rewriting systems [3, 33] covers this subject in-depth.

2.6.1 Term

A *term* is either a *variable* or a *function application*. A function application consists of the function's name and n *operands*, where n matches the *arity* of the function. An operand itself is a term, which is also called *subterm*. In the following, terms are denoted in *function notation* rather than infix notation. In function notation, the function's name is denoted, followed by a tuple of operands. Constants are modeled as *constant functions*, i.e. functions of arity zero. For example, the term $\text{add}(a, 5)$ is an application of the function add on the operands "a" and 5. In this example, "a" is considered a variable and 5 is considered a constant function. Empty parentheses after the function name 5 are omitted for the sake of readability.

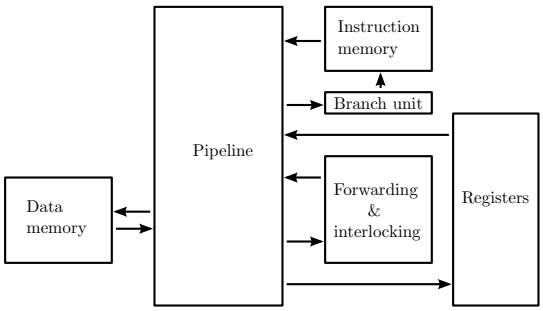


Figure 2.5: Basic structure of a pipeline processor.

2.6.2 Rewrite rules

Rewrite rules define transformations on terms. A rule is denoted $l \to r$, where l is called *redex* and r is called *contractum*. If the redex matches a term, the term can be *rewritten*, i.e. replaced by the contractum. In detail, a redex l matches a term t, if there exists a *substitution* σ that *unifies* l and t. The term is then replaced by $r\sigma$. For instance, the rewrite rule $\text{add}(x, 0) \to x$ can be applied on the term $\text{add}(\text{sub}(3, 2), 0)$. The substitution $\sigma = [x/\text{sub}(3, 2)]$ unifies the redex and the term. The term can be replaced by the contractum $x[x/sub(3, 2)]$, which is $sub(3, 2)$.

Practical rewrite systems are demanded to be *sound*, which means that rewrite rules preserve the meaning of terms. A redex must be *semantically equivalent* to the contractum. As a result, a sound rewrite system only changes the *structure* of a term, but not its *semantics*, i.e. the *interpretation* of the term remains unaffected.

2.6.3 Termination and confluence

A term rewriting system defines a set of rewrite rules. These rules can be used to successively transform a term. If any sequence of applicable rewrite rules is finite for any term, the term rewriting system is said to *terminate*. Termination is an essential property of term rewriting systems. The question if a given term rewriting system terminates is undecidable. A term that can not be rewritten is in *normal form*. If a term rewriting system terminates and each term is rewritten into a unique normal form, the term rewriting system is said to be *confluent*.

2.7 Microarchitecture

The processor generator transforms a ViDL specification into a microarchitectural processor implementation. This section outlines the fundamentals of modern microarchitectures, which are used by the processor generator as described

in Section 7.6. A microarchitecture is a set of concepts and principles used to implement an instruction set. For instance, a microarchitecture may define the structure of a 5-stage pipeline and the concepts of forwarding and interlocking. The ViDL compiler utilizes such principles to produce a processor implementation of a given instruction set specification. This section is intended to give a rough overview and to introduce a common terminology. In-depth information on microarchitectures is given in [24, 4, 41].

A basic microarchitecture of a pipelined processor is shown in Figure 2.5. It consists of a *pipeline*, which executes instructions in a stepwise manner. The pipeline contains the *datapath*, which implements the behavior of instructions. Instructions are *issued* at the top of the pipeline and *committed* at the bottom. *Instruction words* are loaded from the *instruction memory*, which holds the program to be executed. The *branch unit* computes the address of the instruction to be loaded. This is either the successor of the last instruction or the (predicted) target of a branch instruction.

Most instructions read and write registers, while they traverse the pipeline. In long pipelines, this may introduce *data hazards* due to delayed write accesses. Such hazards are resolved by a *forwarding circuit* and by *interlocking*. Interlocking and branches affect the flow of instructions through the pipeline, by means of *canceling* and *stalling*.

2.7.1 Storages

A processor includes a set of different storages. The storages carry the state of the processor. The processor generator has to generate instances of such storages. This section therefore takes a closer look at different kinds of hardware implementations of storages and discusses their properties.

A *register* is a very simple hardware storage, which holds a sequence of bits. It can be read and written in each clock cycle. A one-bit wide register is also called *flag*. The *data in* port of a register receives the next value of the register. The *data out* port yields the current value of the register. A register may provide a one-bit wide *write-enable* port to enable or disable write accesses. If a write access is disabled, the register state remains unchanged. As an alternative to a write-enable bit, a *write-enable mask* may be provided, which allows bitwise enabling and disabling of write accesses. Examples of registers include the *status register*, the *processor mode register* and *special purpose registers*, which are used by exceptional instructions. For instance, a register may be used to store the *processor mode* or the result of some special instructions. The MIPS instruction set includes a pair of *special purpose registers* to store the 64-bit wide result of a multiplication.

A *register file* is an array of registers, which can be accessed *randomly*. To enable multiple concurrent accesses, a register file provides a set of *read-ports* and a set of *write-ports*. The register file can be accesses concurrently through these ports. One read (respectively write) access can be performed through each port at a time. A typical processor may include one register-file of 16 registers, which are 32-bit wide and can be accessed via two read ports and one write

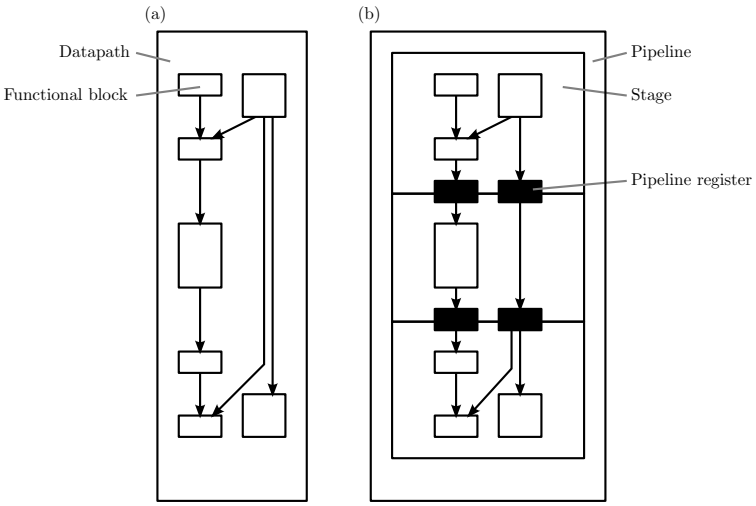


Figure 2.6: Example of a datapath and a pipeline.

port.

Besides registers, a processor typically includes *random access memories* (simply called memories in the following). A memory is an array of *words*. A 32-bit memory for instance consists of a sequence of 32-bit words. The interface of a memory is similar to a register file. However, a memory is larger by some orders of magnitude and slower. The write port of a memory typically includes a (limited) write-enable mask, to allow byte-wise stores.

2.7.2 Datapath

A processor includes a datapath, which basically implements the semantics of all instructions. The processor generator includes sophisticated algorithms to generate such a datapath, as described in Section 7.6. This section therefore gives a short overview of datapaths and their properties.

The *datapath* of a processor consists of *functional blocks*, which are interconnected. The datapath is a directed acyclic graph (DAG), as shown in Figure 2.6a. Each functional block of a datapath has a specific *propagation delay*. For instance, a 32-bit adder may have a propagation delay of 800 ps (picoseconds). This means, that if a value at an input changes, the result is available at the output after 800 ps. The delay of a path is therefore the sum of the propagation delays of all functional blocks on that path. The path with the longest delay is called *critical path*, as it determines the maximal frequency of the processor. A critical path leads from the output of a register to an input of a register. Its delay is also called *register-to-register delay*. It includes the *setup time* and *delay* of the registers.

Figure 2.7 outlines an idealized structure of a typical datapath. The figure

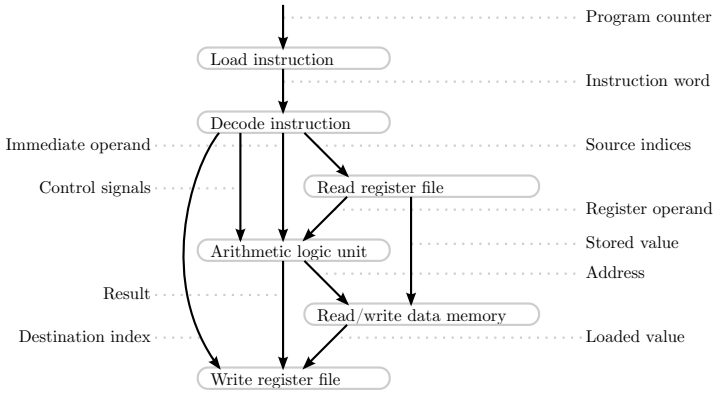


Figure 2.7: Structure of an idealized data path.

is just meant to give an idea of a datapath. Practical data paths are more complex and do not fit into this structure. The processor generator uses a similar structure to generate a microarchitectural processor implementation. At the beginning of the data path, an instruction word is loaded from the instruction memory. The address of the instruction word is given by the program counter. The instruction word is then decoded, yielding control signals, and operands. Control signals are derived from the instruction's *opcode* and used to control the *arithmetic logic unit* (ALU). Operands are either immediate values or indices of source and destination registers. The latter are used to index the register file for *read* and *write* accesses. Immediates and values from the register file are led to the ALU, which computes the result of the instruction. For load and store instructions, the ALU computes the address of the memory access. The computed result (respectively loaded value) is finally stored in the register file.

2.7.3 Pipeline

A direct hardware implementation of a datapath would lead to a slow processor. The processor generator therefore generates a pipelined implementation, to cut down the critical path and thereby the overall register-to-register delay. This section gives a brief overview of pipelines and introduces a respective terminology.

A pipeline consists of a set of *pipeline stages*. The number of pipeline stages is also called *pipeline depth*. The pipeline depth of modern processors ranges from 3 stages to more than 20 stages. The example in Figure 2.6b shows a pipeline of depth 3. The *datapath* of a processor is distributed among the stages of a pipeline. It is intercepted by *pipeline registers*, which are placed between stages. The critical path of the dataflow graph is thereby split into smaller paths. As a result, the maximal *register-to-register delay* is reduced. The processor can be clocked with a higher frequency, which increases *pipeline throughput*. To yield a high clock frequency, propagation delays must be *balanced* among stages, as

indicated in Figure 2.6b.

Stages of a pipeline are typically named by their primary function. A 5-stage pipeline typically consists of the following stages, which implement the functions from Figure 2.7.

IF The *instruction fetch* stage loads an instruction word from instruction memory.

DC/RD The *decode and read* stage decodes the instruction word and loads values from the register file.

EX The *execute* stage computes the result of the instruction.

MA The *memory access* stage loads or stores a value from or to data memory.

WB The *write-back* stage stores the result of the instruction in the register file.

2.7.4 Execution order

For the sake of completeness, the execution order of instructions is briefly described in the following. A simple processor executes instructions *in-order*, i.e. as they appear in instruction memory. The current processor generator produces such an in-order microarchitecture. However, a processor may also use a *dynamic scheduler* to reorder instructions. They are then executed *out-of-order*. The purpose of a dynamic scheduler is to resolve hazards by reordering instructions. For an in-order microarchitecture, this task can be solved by the compiler to some extend. The compiler yields a *static schedule* of instructions, which aims to avoid hazards at execution time. Compared to a dynamic scheduler, a static scheduler does not require any hardware resources, but may yield a worse schedule, as only static program information is available. This thesis regards an in-order microarchitecture, as it requires less resources and power.

2.7.5 Forwarding

To increase instruction throughput, pipelines typically include so called forwarding circuits. To generate forwarding circuits, the processor generator includes sophisticated algorithms, as described in Section 7.6.7. This section therefore gives a brief overview of forwarding and its implementation.

Assume, that a register or register file is read in a stage i and written in a stage $j > i$. Accordingly, these stages are called *read stage* (RD) and *write-back stages* (WB) for this register. Depending on the depth of the pipeline and the datapath, the distance $j - i$ between the read and write-back stage may be large. The result of an instruction may therefore appear delayed by multiple clock cycles. A subsequent instruction does not see the result, as it is not written back yet. This conflict is called a *data hazard* caused by a *read after write (RAW)* dependence. If the result of the instruction is already computed in stage $i+1$, the hazard can be eliminated using a *forwarding circuit*, as shown in Figure 2.8. A forwarding circuit consists of one or multiple bypasses from

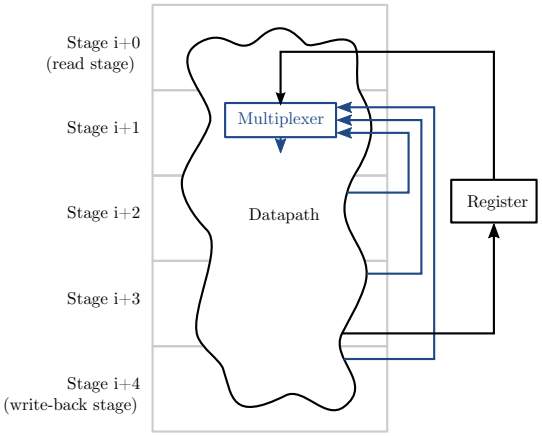


Figure 2.8: Example of a forwarding circuit for one register.

subsequent stages to a *multiplexer*. Each bypass may transport a new state of the register. The multiplexer then selects the most recent state. In the example, the bypass from stage $i+3$ overrides the bypass from stage $i+1$, which overrides the value read from the register.

In a real processor, each register or register file has a separate read stage, write-back stage and forwarding circuit. Hence, speaking about "the" read stage and "the" write-back stage of a pipeline is misleading.

2.7.6 Interlocking

A second technique to eliminate data hazards is interlocking. It is typically applied, if a value can not be forwarded. To generate an interlocked pipeline (Section 7.6.8), the processor generator obeys the principles that are described in this section.

Assume a simple 5-stage pipeline, which consists of the stages IF, RD, EX, MA and WB, as described in Section 2.7.3. Likely, the result of some instructions is not available at the end of EX, but in later stages (MA). Examples include multiplication instructions and load instructions. For such instructions, the result can not be forwarded to a subsequently executed instruction. Instead, the data hazard needs to be resolved by *interlocking*, as shown in Figure 2.9. Basically, interlocking stops the execution of the instruction that accesses the result. In cycle $k+0$ of the example, instruction D accesses a result of instruction C, which is not yet computed and can therefore not be forwarded. As a result, instruction D is *stalled* in stage 1 by interlocking. As a consequence, the instructions in previous stages (i.e. E in stage 0) are stalled too. The instructions in subsequent stages advance as normal. In the resulting gap at stage 2, a so called *bubble* is inserted, which is basically a *no-operation (NOP)* instruction. In the example, instruction D is stalled for two cycles, resulting in two bubbles NOP_1 and NOP_2. In cycle $k+2$, the result of instruction C is finally forwarded

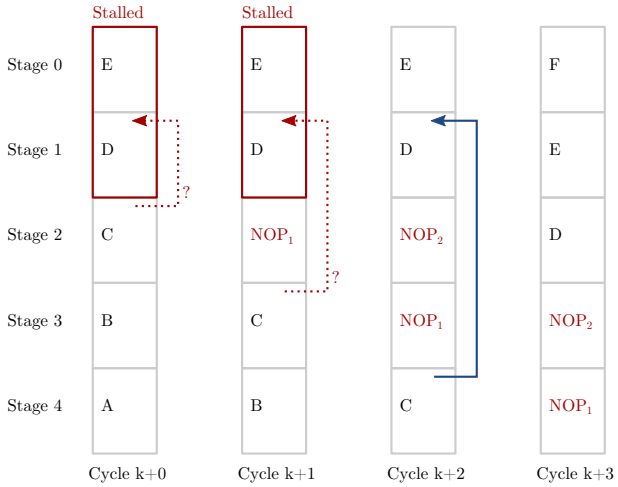


Figure 2.9: Example of two-cycle interlocking in a 5-stage pipeline.

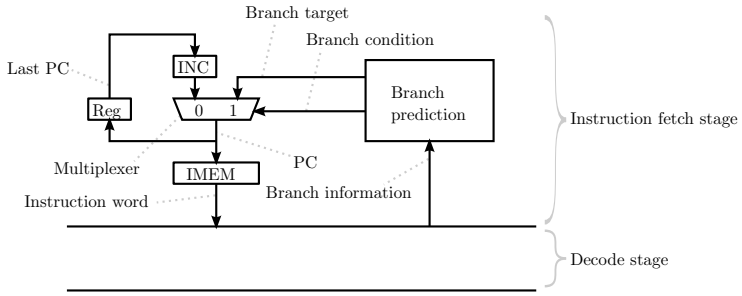


Figure 2.10: Computation of program counter.

to D and execution continues as normal.

2.7.7 Branch prediction

A realistic instruction set includes one or multiple branch instructions. Branch instruction must therefore be implemented by the processor generator. However, branches require a special implementation in a pipelined processor, as they alter control flow. The processor generator therefore includes dedicated methods to generate the implementation of branches. This section presents the foundation for these algorithms, which are discussed in section 7.6.9.

In a processor implementation, a branch instruction changes the control flow by explicitly setting the program counter (PC). The program counter of a processor points to the next instruction to be executed. It is used in the first stage of a pipeline to load an instruction word from the instruction memory (IMEM), as shown in Figure 2.10. If the program counter is not set by a

branch instruction, it is implicitly incremented by the size of the last executed instruction. The program counter is therefore either set to the successor of the last program counter or to the target address of a branch. A branch instruction defines two values, the *branch target* and the *branch condition*. The branch condition corresponds to the condition of a conditional branch. If the condition is true, the branch is said to be *taken* and *not taken* otherwise. An *unconditional branch* (*jump*) is always taken.

A non-delayed branch instructions must virtually set the target value and the condition value, when it enters the decode stage. However, for typical pipelines an instruction is not even decoded in this stage. For a *conditional register branch*, the target and the condition are typically not computed until the memory access stage. To solve this *control hazard*, a pipelined processor includes a *branch prediction*. A branch prediction uses information about a branch instruction to predict the target and the condition. A *static branch prediction* typically regards the direction of a branch to predict the condition. A *dynamic branch prediction* typically considers the address of the branch instruction to predict the target and condition. The probability of a correct prediction depends on the predictor and the executed program. It typically lies in the range of 50% to 98%. In case of a mispredicted branch, *speculatively executed* instructions need to be *canceled*. The number of canceled instructions corresponds to the *branch penalty*.

Chapter 3

Related approaches

The development of processor specification languages started with the developed of modern computers. The "instruction set specification language (ISP)" for instance reaches back to 1970, when it was proposed by Bell and Newell [5]. Since then, a dozen languages for instruction set specification, processor specification and architecture specification have been proposed. Most of these languages use a mixed description of microarchitectural aspects and instruction set aspects, which are strongly coupled. Such specifications are therefore bound to one specific microarchitecture.

ViDL in contrast strictly abstracts from microarchitectural aspects (Section 4.3). As a result, different processor implementations can be generated from the very same specification (Section 1.6). In addition, maintainability and reliability are improved, as microarchitectural aspects need not be considered.

3.1 Taxonomy of ISA specification languages

This section attempts to give a rough overview of processor related specification languages and their characteristics. The languages are discussed comprehensively in the following sections. To give a better overview of existing languages, they are divided into four classes, according to their characteristics. These classes and their relation are illustrated in Figure 3.1 and described in the following. The Figure also shows the degree of abstraction and the expressiveness of each class, as these are significant properties in practice.

HDL The class of *hardware description languages (HDL)* includes languages like VHDL or Verilog. In such languages, a processor is defined on a microarchitectural level (Section 1.3). Any detail of the hardware processor is defined, including the structure of pipelines, functional units and their control. Distinct instructions are not apparent. Instead the semantics of all instructions are merged and distributed over a variety of resources. For instance, an add-instruction contributes to the decoder, the ALU, for-

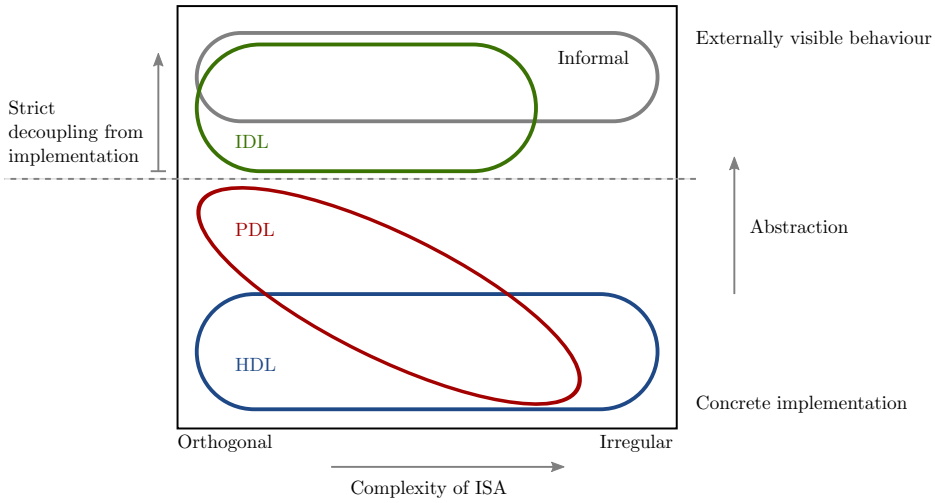


Figure 3.1: Abstraction and expressiveness of processor related specification languages.

warding, interlocking and register write-back. The ALU on the other hand implements the functionality of all arithmetic and logical instructions.

Compared to processor and instruction set specification languages, the level of abstraction is low. The specification is typically complex and requires extensive testing. On the other hand, hardware description languages are very powerful. They can be used to define any kind of processor in terms of instruction set and microarchitecture. The space of processors that can be specified is not limited by an underlying processor model.

PDL Most processor related specification languages belong to the class of *processor description languages (PDL)*. This includes nML (3.4), Lisa (3.6) and Expression (3.8). Processor description languages define aspects of the instruction set, as well as aspects of the microarchitecture. The definition of microarchitectural aspects is essential for PDLs, i.e. they can not be omitted. As a result, a PDL does not only define an instruction set, but actually a specific hardware implementation of a processor. This thesis therefore uses the term "processor description language" for such languages rather than "instruction set description language".

In a PDL, the microarchitecture is specified according to a language specific model. The model greatly affects the expressiveness of the language. For instance, a model may not be able to represent conditional execution or delayed instructions. Compared to an HDL, a PDL imposes certain restrictions, i.e. it may not be possible to specify some processors in a natural way.

PDLs define an instruction set on the foundation of the specified microar-

chitecture. The instruction set description is thereby bound to that specific microarchitecture. The microarchitecture can hardly be changed, once the instruction set is defined.

Using a PDL, a developer likely has to specify pipeline control manually. For instance, stages must be stalled and flushed, such that all data hazards and control hazards are correctly resolved. This task is quite complex and extensive testing of the resulting processor implementation is therefore advisable.

A PDL specification is translated into HDL code using a generator. Some generators are not fully automated and require manual intervention or the specification of additional HDL code. The implementation effort of most generators may be considered low, since the structure of most PDL specifications is quite close to that of HDL specifications. In-depth information on the implementation of generators is not available in literature.

IDL The class of *instruction set description languages (IDL)* includes the specification languages ViDL (4) and TIE (3.9). An IDL specification focuses on the aspects of instruction sets, such as programmer accessible storages and semantics of instructions. An IDL abstracts from microarchitectural aspects, similar to most instruction set manual. For instance, the pipeline structure, branch prediction and forwarding are not defined. The abstraction from microarchitecture enables the generation of very different processor implementations, using the same instruction set specification. Compared to a PDL generator, an IDL generator is much more complex. It has to derive many aspects of the microarchitecture by analyzing instruction semantics. The analysis methods of ViDL's generators are discussed in Chapter 7.

Informal *Informal texts* or *semi-formal texts* are for instance used in manuals 3.2 to describe instruction sets. Any concept of an instruction set can be described in an informal natural language. However, informal texts can not be processed by generators. An instruction set manual is for instance not suited as input to generate a processor implementation. Nevertheless, it is worth taking a look at manuals, as they describe instruction sets in an intuitive and simple manner. The structure of ViDL is actually very similar to manuals for this reason.

3.2 Notation in ISA manuals

Most instruction set manuals define the semantics of instructions and the structure of storages in a semi-formal way. Instruction semantics are likely denoted using some sort of pseudo-code. Basic operations in the pseudo code like "+" and "<" (also known as pragmatics [20]) are assumed to be self-explanatory or are described informally by a text. Such descriptions are necessary to interpret the pseudo-code.

The purpose of a manual is to describe instruction semantics for humans. It is not intended as formal input to generators. In the following, the ARM manual is regarded. Other manuals, such as the Power manual, the MIPS manual and the CoreVA manual have a similar structure.

3.2.1 ARM manual

In the *ARM architecture manual* [2], instruction semantics are described semi-formal in terms of pseudo code. The pseudo code is defined in an imperative language. Registers and memories are modeled as scalar variables and arrays. Instruction semantics are defined using a set of primitive operations, such as `SignExtend` and `Logical_Shift_Right`. The semantics of primitives are described informally in a glossary. Common semantics of instructions are factored out. The addressing modes of load and store instructions are for instance defined once and then referred in the descriptive text of each instruction.

The state of the processor, which consists of registers and the main memory, is informally defined in the ISA manual. The ARM architecture uses banked registers, which means that different sets of physical registers are associated with register names, depending on the current processor mode. For example, the register name `R14` is associated with the physical register `R14_svc` in supervisor mode and with `R14_fiq` in fast interrupt mode. This association is visualized and explained by informal text.

The encoding of each instruction is defined by diagram, which divides the instruction word into named fields and opcode bits. The field names are then used in the specification of instruction semantics, to refer to the respective bit-slices.

The memory is modeled as an array. An access to the memory is denoted `Memory[<address>,<size>]`, where `<size>` is the size of the memory access, which is 1, 2 and 4 bytes for ARM.

3.2.2 Review

The description of instruction semantics in manuals is clear. Each instruction can be regarded separately by the reader. Common aspects are factored out, to simplify the specification due to reuse. In the same way, instructions are specified in ViDL, i.e. each instruction is defined on its own. Common aspects are factored out using functions and functional. The encoding is defined in ViDL using a pattern, similar to manuals.

The notation of memory accesses in manuals is intuitive and simple. It encapsulates masking, write-enabling and endianness. A similar notation can be used in ViDL, by defining so called views for byte-, half-word- and word-accesses. Manuals describe semantics of instructions on a high level, using high level primitives, such as “rotate” or “sign extend”. The same holds for ViDL, which defines such primitives in an extensible library. Both, manuals and ViDL focus on the instruction set. They strictly abstract from microarchitectural

aspects, such as the pipeline structure or register ports. The description is thereby simplified and it is not bound to one specific processor implementation.

3.3 ISP

The specification language *ISP* [5] (Instruction Set Processor) was first used in the Book "Computer structures: readings and examples" [6] of Bell and Newell in 1971. This language was intended to formally describe the behavior of processor architectures. The notation was adopted by the SPARC manual [48] to formally describe instruction semantics. Basically, the language targets at the interpretation of binary programs and the generation of hardware.

3.3.1 State

The processor state is described as a set of storages. A storage models for example a register, a register file or a memory. Conceptually, each storage represents a bit-string of a specific width or an array of a specific size. The elements of the array are either bit-strings of a specific width or are itself arrays. The latter can be used to model multi-dimensional storages.

3.3.2 Aliases

ISP provides aliases to define alternative names for bit-slices of storages. The names can be used to refer to these bit-slices. Assume for instance a 32-bit accumulator register. Two 16-bit aliases may be defined in ISP to refer to the high and low part of the accumulator. An alias in ISP is a special case of an architectural interface in ViDL, but is more restricted. An alias always refers to one specific storage element. For instance, a 16-bit view on a 32-bit register file can not be defined. Aliases are static, i.e. they can not be changed or reconfigured dynamically. As a result, reconfigurable register structures, such as mode-dependent registers, can not be defined.

3.3.3 Instruction encoding

The encoding of instructions in ISP is defined globally for all instructions. Aliases are used to refer to bit-slices of the instruction word, as shown in the following example.

```
x<0:1> := instruction<0:1>
y<0:1> := instruction<2:3>
opcode<0:4> := instruction<4:7>
```

The identifier `opcode` refers to bits 7 to 4 of the instruction word. It is used later in the definition of instruction semantics, to identify instructions. The encoding of a specific instruction is therefore not obvious, as it is defined by semantics.

3.3.4 Activations

In ISP, the association between instructions and opcodes is expressed by so called *activations*. An activation is an expression, which triggers the "execution" of an instruction. Activations typically compare the opcode bits of an instruction word to constants. If the comparison matches, the activation holds and the corresponding action is triggered.

The concept of activations is very close to a straight forward implementation of a hardware decoder. The decoder concurrently evaluates all activations and yields one signal for each activation. This results in n signals for n instructions, where the i-th signal is active, if the i-th instruction is executed.

The concept of activations is therefore quite hardware centric and implies a certain implementation of the decoder. It is suited for describing the semantics of a processor and for generating hardware, but generating efficient software decoders (as found in simulators) may be considered hard. In software, the activations are evaluated sequentially and not in parallel as in hardware. The execution time of the software decoder is therefore expected to be linear in the number of instructions. The use of patterns in ViDL allows generating a concurrent hardware decoder and a decision tree based software decoder. The execution time of the decoder is typically logarithmic in the number of instructions.

Besides, the concept of activations is not well suited for generating compiler tools. An assembler for instance needs to construct an instruction word for a given instruction with its operands. This task is basically the inverse of decoding and would require determining the inverse of all activation expressions, which is not trivial. Compared to that, the derivation of an encoder is quite simple for ViDL, as a pattern implies a bidirectional mapping between operands and bits of the instruction word.

For the typical one-to-one relation between activations and instructions, activations typically need to be disjoint. This means, that there must not be two activations that hold for the same instruction word. The developer therefore needs to ensure, that activations are disjoint, which is not a trivial task. This is especially true, if instructions are not only identified by a single opcode field. If the instruction encoding is specified by patterns instead, the generator can check for disjoint instructions by very simple and fast operations on the patterns.

3.3.5 Actions

The semantics of instructions are specified by so called *actions* in ISP. An action is a sequence of assignments, which are executed concurrently, as long as there is no intervening `next` statement. This is similar to the paradigm of concurrent assignments in VHDL.

In ISP, primitive operations like "+", "×", "∧", "⊕" are used to specify the semantics of instructions. The set of available operations and their precise semantics are not defined in literature.

3.3.6 Data-types

In ISP, bit-strings may be interpreted in different ways. For example, a string of 32 bits may be interpreted as a floating point number with single precision or as an unsigned integer. The particular interpretation of bit strings is specified explicitly on expression level. In detail, one interpretation can be specified for each side of an assignment. All operands on each side are then interpreted the same way.

If the interpretations on the two sides are different, the expression on the right hand side is converted to the interpretation of the left hand side. The semantics of the conversion can not be defined and there seems to be no way, to influence the conversion. There are for example at least 3 meaningful ways to convert a floating point number into an integral number: Rounding to the next smaller integer, rounding to the next larger integer or rounding towards zero.

To specify arithmetic operations on bit-strings, which are interpreted as floating point numbers, primitives are overloaded. The arithmetic primitive "+" is for example overloaded for the interpretations "integer" and "floating point". It is left open, which operations are overloaded for which data types and what their semantics are. For example, if there are meaningful semantics for the $\oplus$ primitive on floating point numbers.

ISP suggests a set of interpretations of bit-strings, including a three letter scheme for denoting floating point interpretations. Beyond this set, there is an infinite number of meaningful interpretations, which are not covered by the proposed set of interpretations.

To overcome the complexity introduced by multiple data types, ViDL uses only one class of data types namely *bit-string*. The interpretation of a bit string as integer or floating point is folded into primitives. There may for example be one "add-integer" primitive and one "add-ieee754-single" primitive. A primitive is therefore not overloaded for different interpretations.

3.3.7 Review

Simplicity ISP is a quite simple ISA specification language. It uses only a limited number of concepts to describe ISAs. The set of instructions is defined implicitly by a hierarchy of operations, activations and sub-operations. Basically, this is a decision tree on the encoding of instructions. The nodes of the decision tree are annotated with assignments, which define the next processor state. This structure is different from instruction set manuals, where each element of the instruction set is defined separately. Formalizing an instruction set manual in ISP may therefore be complex, as the structure of the instruction set must be analyzed first by the developer.

Maintainability The structure of the instruction space and the semantics of instructions are strongly coupled by the decision tree. This has a major impact on maintainability and may complicate design space explorations and instruction

set extensions. Instructions can not directly be added or removed from the set of instructions.

Expressiveness Compared to other approaches, the expressiveness of ISP is limited. The language was designed in the 70s and is suited for simple instruction sets without visible pipeline effects. Properties of modern instruction sets, such as delay slots, conditional execution or register windowing can not be described. Only a very basic set of arithmetic and logic operations is used in [5] to define a PDP-8 instruction set. An approach to formally specify further operations is not mentioned. The width of operations and the analysis of bit-widths are left open. ISP provides the concept of aliases. An alias defines a static view on a single register. It is a special case of architectural interfaces in ViDL (Section 4.8).

An ISP specification is founded on a decision tree. This tree imposes a certain structure on the instruction set. It is well suited to develop highly orthogonal instruction sets from scratch. However, existing instruction sets that do not exactly fit this structure may be hard to describe.

Reuse Aliases in ISP offer a kind of reuse. A view on a bit-slice is defined once and then reused among the specification. In addition, ViDL also offers further concepts to enable reuse and factorization. This includes variables, functions and functionals.

Reliability As assignments are distributed over the decision tree, the effects of an instruction may not always be clear. Unintended side-effects may result, which lead to incorrect specifications. To reason about the effects of an instruction, great parts of the specification must be understood. This is different in ViDL, where each instruction is specified on its own. An instruction in ViDL is self contained, except for applications of functions. However, functions do not have side-effects, which need to be considered.

3.4 nML

The specification language *nML* [15, 17, 44] belongs to the class of processor description languages (PDL), as aspects of the microarchitectural implementation are defined. An nML specification includes for instance a definition of the pipeline structure and is thereby bound to a specific microarchitecture. The concepts of nML are well suited to develop highly orthogonal instruction sets from scratch. The specification is founded on a so called "structural skeleton", which reflects the structure of the instruction set as well as the structure of instructions. The skeleton can be used to factor common aspects of instructions out. This concept is described later in detail.

An nML specification can be divided into the specification of the processor state (Section 3.4.1) and the specification of instructions (Section 3.4.2). All instructions are specified commonly using the formalism of attribute grammars. The elements of the instruction set are therefore not enumerated explicitly.

Section 3.4.2.1 takes a very close look at this formalism and discusses open questions and inaccuracies. It should be noted in advance, that this discussion is based on a very limited amount of available literature on nML. The correctness of the following conclusions can therefore not be guaranteed. The section closes with a brief discussion of instruction set modeling (Section 3.4.3) and aspects of language quality (Section 3.4.4).

3.4.1 State

In nML, the state of a processor is defined by a set of storages. Each storage is defined by a number of elements and the width of elements. A storage may for example be a register-bank of 8 registers, where each register is 64-bit wide. Using these storages, all memories and registers of the ISA are defined. Basically, these are all storages that are directly exposed to the application programmer and therefore described in the ISA manual. In addition, microarchitectural storages, such as pipeline registers are defined in nML. These storages are not part of the ISA. The number and size of these registers depends on the chosen microarchitecture and may be different for different implementations of the same ISA. For a non-pipelined implementation of the ISA, the processor implementation will not contain pipeline registers at all. The nML specification does therefore not only specify the ISA, but also aspects of the microarchitecture, like the number of pipeline stages. Changing the microarchitecture at a later time may therefore necessitate expensive modifications of the processor specification.

3.4.2 Instruction set

An nML specification defines a set of instructions in a structural way. Instead of defining each instruction separately, a single structure is defined, from which all instructions are derived.

Instructions and their semantics are defined using so called *rules*. The set of rules implies a set of *trees*, where each tree corresponds to one instruction. A node of a tree is called an *operation*. Operations contribute the behavior, encoding and assembly syntax of an instruction.

The *set* of instructions is thereby defined intensionally[1] in nML. It is derived from the set of rules and not specified extensionally, as in ViDL. An intensional definition is well suited for highly orthogonal instruction sets. Common aspects of instructions can be factored out into common operations. Exceptional behavior however is hard to express in nML, as orthogonality and the respective factorization are fundamental concepts in nML. In contrast to nML, ViDL defines each instruction independently. Common behavior may be factored out using functions or functionals. However, factorization is optional in ViDL and can be applied by the developer as desired. For instance, the specification of a small irregular instruction set may not use any factorization at all. The definition of each instruction is therefore self-contained.

[1] In terms of intensionally defined sets in mathematics.

3.4.2.1 Attribute grammar

The formalism behinds an nML specification is basically an attribute grammar [34] that obeys a certain structure. An attribute grammar $AG = (G, A, C)$ is defined by a context free grammar G, a set of attributes A and a set of computations C. A context free grammar G is defined by a quadruple $G = (T, N, P, S)$, where T is the set of terminal symbols, N is the set of non-terminal symbols, P is the set of productions and S is the start symbol.

In nML, the set of terminals is empty ($T = \emptyset$), as every non-terminal appears on the left hand side of exactly one production. A non-terminal $x \in N$ corresponds to an operation in nML. A production $p \in P$ corresponds to a rule in nML. A rule must either be a so called AND-rule or a so called OR-rule. An OR-rule corresponds to a production of the form `X ::= X1 | X2 | ... | Xn` and an AND-rule to a production of the form `X ::= X1 X2 ... Xn`. The set of valid productions is thereby restricted. For instance, the production `X ::= X1 X2 | X3` does not model a valid nML Rule. For each non-terminal `X`, there must be exactly one production of the form `X := ...` in P. As a result of this constraint $|P| = |N|$ holds.

Each non-terminal has exactly three synthesized attributes, namely `ACTION`, `IMAGE` and `SYNTAX`. The `ACTION` attribute defines the behavior of an operation. The `IMAGE` attribute defines its encoding, i.e. how an operation is encoded as part of an instruction word. The last attribute `SYNTAX` defines the assembly notation of the operation, i.e. how it is represented as part of an assembly instruction. All attributes are synthesized, i.e. they can be computed during a single bottom up pass in the tree. An nML specification therefore belongs to the grammar class SAG.

3.4.2.2 Grammar properties

The nML grammar defines an *abstract syntax*, i.e. it defines a set of trees. The language of such a grammar contains at most the empty word, since the set of terminals is empty ($T = \emptyset$). Hence, the set of instructions corresponds to the set of derivation trees and not to the language $L(G)$ as stated in [44].

An nML grammar may not be sound, as demonstrated by the following example.

```
A ::= B | C
B ::= A C
C ::=
```

According to nML literature, the grammar is valid. The first production corresponds to an OR-Rule and the second production to an AND-rule. However, this grammar implies an infinite set of derivation trees and thereby an infinite set of instructions. As a result, the number of `A`, `B` and `C` nodes in a derivation is unlimited. Depending on the definition of attributes, this may lead to instruction encodings of unlimited width and data-paths of unlimited length. The specification is therefore not sound.

Although not stated in nML literature, it seems necessary to demand non-recursive rules to ensure soundness. Formally this means, that there is an ordering "<" on the set of non-terminals, such that the symbol on the left hand side of a production is smaller than the symbols on the right hand side. A developer should be aware of this restriction, to define sound instruction sets.

3.4.3 Modeling of instruction sets

In nML, OR rules partition a set of instructions into disjoint instruction sets. An AND rule specifies, that all instructions in the set of instructions are composed of multiple disjoint sub-operations. For an existing instruction set description (e.g. an ISA manual), it may be hard to derive the corresponding nML grammar. Common instruction aspects and orthogonal sub-instructions need to be factored out.

In the process of design space exploration, instructions are typically added, changed or removed. In nML, instructions are coupled via operations. Changing the semantics of only one single instruction may therefore be difficult. For instance, to modify the semantics of an instruction, an operation needs to be changed, which may then affect other instructions.

An instruction corresponds to a derivation tree, which represents a dataflow graph. Each node (operation) in the tree corresponds to one logical block. Data flows from child nodes to parent nodes. The `ACTION` attribute defines the transfer function of the logical block. The ports of a node or logical block are not defined explicitly in nML. Instead, global signals are used in the specification to communicate between logical blocks. The designer needs to take care, that the dataflow specified using global signals matches the structure of the derivation tree.

In nML, a single structure (the grammar) is used to specify the semantics, the encoding and the assembly notation of instructions. It is assumed, that all aspects obey the same structure. This may be a problem, if the instruction set structure differs for the three aspects. Assume for instance, that the instruction pairs A/B and C/D have common semantics, but the pairs A/C and B/D have a common encoding.

3.4.4 Review

Abstraction Parts of an nML specification are defined in C++. This includes the definition of element types and the address types of storages. The embedding of C++ code breaks abstraction from target languages. As a result, the implementation language of a simulator is bound to C++. For instance, the language is not suited to generate a simulator in terms of Java code.

Great parts of the microarchitecture are explicitly specified in nML. This includes signals, functional units, register ports and the instruction pipeline. The latter includes pipeline registers and pipeline control. An nML specification is thereby bound to one specific microarchitecture. Due to the lack of abstraction, it is virtually impossible to generate different microarchitectural

implementations from the same specification. Aspects of the microarchitecture are the foundation of an nML specification and can not be selected at a later time.

Literature states, that the generator GO generates a netlist of RTL blocks from an nML specification. The implementation of these blocks in terms of HDL code is not mentioned. It is also not stated, if the code is ready for synthesis or if certain parts need to be contributed manually by the developer.

Expressiveness As nML does not strictly abstract from the target domain, it can cover a wide range of processors. The pipeline is under the developer's control, which enables uncommon and irregular timing behavior.

An nML specification uses a common "structural skeleton" to specify the behavior, assembly notation and encoding of instructions. Instructions with a common encoding are assumed to also share a common behavior and a common assembly syntax. Instruction sets that do not conform to this condition are not discussed in literature.

Simplicity The language defines a variety of syntactic elements and concepts that are tailored to the domain of processor specification. This section has only focused on a small fraction of nML and its language constructs. The instruction set structure is defined using a grammar in nML. Users in the domain of processor development may not be familiar with this formalism. A robust understanding of grammars and their properties may be necessary to specify real world instruction sets. Section 3.4.2 has demonstrated the complexity and pitfalls of this formalism.

Significant parts of an nML specification are defined by embedded C++ code. This includes the specification of data types and primitive functions. Therefore, users likely need to know the programming language C++ to some extend. Since data types are defined in C++, nML uses the type-system of C++. Implicit typing by type inference or generic bit-string types are not mentioned in literature. Instead, it seems necessary that instruction set specific types and respective conversion functions must be specified in C++. The specification appears to be typed explicitly. Both increase the complexity of a specification.

To formalize an existing instruction set manual, a developer needs to analyze the instruction set. Common behavior has to be factor out and appropriate instruction groups need to be identified. This is a complex process of reverse engineering, where a set of derivation trees (i.e. instructions) is given and an appropriate grammar must be reconstructed.

Reuse Reuse is provided by the structural skeleton. Each operation in that skeleton may contribute to multiple instructions. Further concepts of reuse, such as abstraction by functions, are not mentioned. Architectural registers can be defined in terms of storage aliases. The mapping must be static, uniform and registers must have the same size, i.e. the same type in nML. Virtual

address spaces or mode specific registers can not be modeled by aliases. Such ISA properties must be specified as part of the semantics.

Reliability Pipeline control is specified explicitly by the developer, which is typically a complex and error-prone task. Bypasses, interlocking, stalls, flushes and delay slots must be considered properly.

The behavior of an instruction is constituted by the respective "ACTION" attributes. The attributes are distributed over the ISA grammar and the effects of an instruction may therefore not immediately be clear. In particular, unintended side effects may lead to errors.

Maintainability Consistent changes to a group of related instructions can efficiently be performed, by changing the attributes of the common non-terminal. Changes that affect the instruction set structure seem to be complex. Unfortunately, this case is not discussed in literature. As the grammar couples the encoding, behavior and syntax of instructions, these aspects can not be changed independently. Moving an instruction to a different location in the instruction space will for instance also affects its behavior.

3.5 ASIP Meister/PEAS-III

A specification in the language *ASIP Meister*[29] (formerly known as PEAS-III) basically describes a processor on a microarchitectural level. Pipeline structures, such as stages and storage ports are specified explicitly. The language therefore belongs to the class of processor description languages (PDL).

The behavior of instructions is defined by micro-operations, which are similar to operations in nML. A micro-operation is defined in the "Micro-operation description language", which only consists of (conditional) signal assignments and applications of so called resource-functions. Resources and their functions are defined externally in a database called FHM-DBMS. Putting all together, the behavior of instructions is composed from the behavior of resources. The specification of resource behavior is therefore a crucial component of the approach. Unfortunately, literature on FHM-DBMS and its specification language appears to be unavailable. Only the popular "add" example (`c=a+b`) is presented in [35]. However, the language behind this statement is not described. It remains unclear, how a complex function, such as "count-significant-bits" is formally specified in FHM-DMBS.

Review

Reuse Similar to primitives in ViDL, resources provide a way of reuse, as they abstract from processor specifications. Resources are coarse grained building blocks (ALUs), which encapsulate multiple functions. They are intended to be utilized by multiple instructions. Resources enable explicit sharing of hardware among multiple functions.

In contrast, primitives in ViDL enable a fine-grained specification of instructions. Each primitive provides only one basic function, such as "add" or "count-leading-zeros". Complex behavior is then defined in ViDL by combining primitives to instructions or functions. As primitives are less customized than resources, one may expect a higher degree of reuse.

Another kind of reuse in ASIP Meister is provided by micro-operations. A micro-operation is defined once and then reused among multiple functions. General concepts for reuse, such as functions or functionals are not mentioned in literature.

Expressiveness The resolution of data-hazards by means of bypasses and interlocking seems not to be covered by ASIP Meister. Peddersen et al. [43] describe, how to patch generated HDL code (from ASIP Meister) to include register bypasses. The approach necessitates the specification of additional pipeline registers and signals, to control forwarding. A concrete algorithm to generate bypasses is not presented and it remains unclear, to which extend the approach can be applied.

Simplicity Signals are typed explicitly by declarations such as `wire[32:0]`. Type inference does not seem to be applied. Literature does not state explicitly if the width is arbitrary or limited in some way. However, it is noticeable, that examples in literature do not exceed 32 bits and that the resource in [35] explicitly declares parameters to be of type `int32`.

3.6 Lisa

The specification language *Lisa* is very similar to nML. A Lisa specification defines aspects of the instruction set as well as aspects of the microarchitecture. It is therefore bound to one specific microarchitecture. The instruction set is specified on the foundation of this microarchitecture. Lisa is therefore considered a processor description language (PDL) and not an instruction set description language (IDL).

3.6.1 Storages

Similar to nML, a Lisa specification defines a set of storages, where each storage is characterized by its size and its width.

The choice of width is limited for some cases. For instance, the width of the program counter register must correspond to the width of a C integer type[2]. A 30-bit program counter can for instance not be defined in Lisa. To generate an efficient simulator, the same restriction must be obeyed by all registers. For instance, according to the Lisa manual [27], the definition of a 48-bit wide registers would significantly slow down the generated simulator.

[2] `stdint.h` defines 8,16,32 and 64-bit integers

3.6.1.1 Data-types

The definition of a storage also includes the definition of its interpretation. A storage is defined to be either signed or unsigned. This choice basically controls the extension of the registers content (zero or sign). This aspect should not be bound to a storage, as it belongs to instruction semantics, rather than to the processor state. Different instructions may for instance interpret the content of the same register differently. Wide multiplications of MIPS interpret the content of as a signed (`mult`) or unsigned integer (`multu`).

A storage in Lisa is always defined to contain a signed or unsigned integer. However, there are various different data-types, which imply other interpretations. For instance, a register may contain a sequence of flags (status register), a floating point number or an integral number in sign-magnitude representation. An implicit sign or zero extension of such values, as defined in Lisa, is not sound.

3.6.1.2 Pipeline registers

To generate a pipelined processor, pipeline registers must be defined in Lisa. This is a result of the mixed ISA/microarchitecture paradigm. The definition includes the width and the stage of each pipeline register. The pipeline registers are then used in the specification of instruction semantics. Instruction semantics are thereby divided and effectively bound to pipeline stages.

As a result, the microarchitecture can hardly be changed after defining the instruction set. This restriction may limit the application of Lisa in the scenario of design space exploration. Effectively, the instruction set architecture and the microarchitecture are strongly coupled.

3.6.2 Instruction set

In Lisa, an instruction is constituted by a set of *operations*. Each operation has three attributes, which define the semantics, encoding and assembly syntax. The semantics, encoding and syntax of an instruction is then composed from the attributes of the operations.

This approach is similar to nML, where the same attributes are defined for operations. A major difference between nML and Lisa is the specification of the relation between operations and instructions. In nML the set of instructions and their relation to operations is defined by a grammar. In Lisa, the relation is defined explicitly by listing the operations of each instruction.

3.6.2.1 Semantics of operations

The semantics of operations may be defined redundantly in Lisa. Once in terms of C-code for the simulator and once for the generation of synthesizable HDL code. The developer must take care to define exactly the same semantics in both definitions. Otherwise, the behavior of the simulator may not be consistent to the hardware processor's behavior. Extensive equivalence testing of both generated products is therefore advisable.

3.6.2.2 Errors in semantics

Using Lisa, the semantics of an operation can be defined in terms of C-code. It seems that these code fragments are literally copied into the source code of a simulator without further analysis or transformations. Muhammad et al. [40] report, that syntactic errors in the Lisa specification are not detected by the Lisa generator. Instead, incorrect simulator code is generated, which is then rejected by the C-compiler. Identifying such an error in the Lisa specification may be time-consuming.

3.6.2.3 C-to-VHDL translation

The authors of Lisa state [7], that the C-code of the behavioral specification can be translated into HDL code. Restrictions or limitations are not mentioned. However, such restrictions actually exist. Only a very limited subset of C-code can be translated into VHDL code by the generator. For instance, typical C-code that includes loops or function calls can not be translated into HDL code. In such a case, the generator reports an error and aborts translation. The separate specification of behavior for the simulator and processor appears to be necessary. It has been reported by users from industry, that two independent Lisa specifications have been developed, one for the simulator and another one for the hardware model. Similar to the independent development of simulator and hardware, this scenario is laborious and requires extensive testing for equivalence.

Even if the Lisa generator would support translation of arbitrary C-code into HDL code, it is questionable that the resulting HDL code is efficient. Regard for instance the following C-implementation of a well known processor instruction.

```
for (i=0; i<32 ; ++i){
  if ((a>>(31-i))&1) break;
}
res=i;
```

Even for the experienced user, it may not be obvious that this code is a straight forward implementation of the count-leading-zeros instruction for 32 bit processors. There are at least a dozen alternative ways to describe the same semantics in terms of C-Code. A good C-to-HDL translator would have to identify all these implementations and map them to an efficient HDL implementation, i.e. to a priority decoder.

3.6.2.4 Wide operations

The semantics of instructions can efficiently be defined in C, as long as the width of operations matches the width of C-integer arithmetic, i.e. 8, 16, 32 or 64 bit. If the width of an operation does not match one of these widths, the C-code tends to get very complex and error-prone. Narrow operations require explicit masking in C and wide operations must be broken down to sequences of C-operations.

For example, a 96-bit unsigned multiplication must be reduced to a sequence of multiplications, additions and appropriate masking. In the following example, `a.lo, a.hi, b.lo, b.hi` are assumed to be 64 bit wide integer variables.

```
res.lo  = a.lo*b.lo;
res.hi  = ((((a.lo>>32)*b.lo&0xffffffff) >> 32)
         + (((b.lo>>32)*a.lo&0xffffffff) >> 32)
         + (a.lo>>32)*(b.lo>>32);
         + a.hi*b.hi) & 0xffffffff;
```

A good C-to-HDL translator would have to recognize this code as a 96 bit wide multiplication and generate a 96 bit wide instance of a multiplier component. If the pattern of the 96 bit wide multiplication is not recognized, the C-to-VHDL translator is likely to yield an expensive cascade of 5 64-bit multipliers and 3 64-bit adders.

As long as Lisa does not include a C-to-HDL translator of exceptional quality, either the generated VHDL code is poor or the semantics must be specified redundantly, which leads to potential inconsistencies and thereby breaks the claim of a golden model.

3.6.3 Hardware sharing

The sharing of hardware is specified explicitly in Lisa, in terms of operations and units. To share the same hardware entity, two instructions are specified to use the same operation. For instance, addressing modes are specified once as operations and then used by a set of instructions. The instructions do thereby share the logic for decoding of operands and calculation of addresses.

Another way to explicitly share resources is the use of so called “units”. Units are only used in the context of HDL code generation, i.e. they do not apply to simulators. The processor designer explicitly maps a set of operations to one unit. The unit then implements the semantics of all these operations. For instance, the operations “Add”, “Sub”, “Mult” may be mapped to a unit “ALU”. The ALU must then be specified such, that it unites the semantics of the three operations. In detail, a HDL source-code frame is generated for each unit. This frame must then manually be filled with appropriate HDL code, depending on the set of mapped operations.

This is a critical point in processor development, as the specification of operation semantics must match the semantics of the unit’s HDL code. Otherwise, the simulator and the processor implementation would be inconsistent. Once the unit is implemented, changes to operations imply consistent changes to the unit.

ViDL strictly abstracts from microarchitecture and consequently from hardware sharing. Hardware sharing is postponed to the generator, such that user can focus on ISA development. Besides, instructions remain independent in ViDL, as they are not coupled by shared resources.

3.6.4 Pipeline

Pipelining has to be specified explicitly in Lisa. The specification includes the number and names of pipeline stages. These names are then used to manually assign operations to specific stages.

In contrast, the pipeline depth is not specified in ViDL, but derived automatically by the ViDL compiler. The pipeline depth and structure is a microarchitectural aspect, which does not belong to an instruction set. It is therefore not specified in ViDL, which strictly abstracts from microarchitecture.

3.6.4.1 Forwarding

In Lisa, a forwarding circuit is defined on a microarchitectural level, as it is done in a VHDL specification of a processor. To forward a value to early pipeline stages, the developer has to define bypass signals. These signals are then "written" by a late stage and "read" by an early stage. The definition of a typical forwarding circuit in Lisa may be considered complex. For instance, a very simple 5-stage MIPS pipeline includes 6 bypasses [24] for the general purpose register file.

Each bypass consists of 3 signals, namely the forwarded value, the register number and the activation signal. In total, this makes 18 signals. These signals are fed into a special multiplexer, along with values from the read stage. Note, that the multiplexer is quite complex, as it must consider register numbers, activation signals and the number of source stages. It seems that the multiplexer logic must be defined manually in Lisa. This task may be considered complex. Although forwarding is a key concept of pipelines, a respective example is not given in literature.

A ViDL developer does not have to consider forwarding, as the forwarding circuit is automatically contributed by the generator. This greatly simplifies the instruction set specification and avoids flaws. The generated forwarding circuit is guaranteed to be correct and consistent. In addition, processor implementations with different pipeline structures and forwarding circuits can be generated from the very same ViDL specification. This allows for an automated exploration of the microarchitectural design space.

3.6.4.2 Pipeline control

Realistic processors typically involve hazards, due to pipelining. Such pipeline hazards have to be resolved by respective pipeline control. In particular, stages have to be stalled and flushed. A Lisa developer has to define this pipeline control manually, as part of operation semantics. In case of a mispredicted branch, all speculatively executed instructions have to be canceled, by flushing their stages. If a register is read, that is written by a preceding instruction X and the result of X can not be forwarded due to its latency, the read stage and its predecessors have to be stalled, until the result is computed and can be forwarded. The designer is responsible to define correct pipeline control that

solves all hazards for all sequences of instructions. This task can be considered complex and error-prone.

Fortunately, this task is solved by ViDL's processor generator. A ViDL developer does not have to consider hazard and their resolution. The specification is thereby simplified and a major source of processor flaws is eliminated.

3.6.5 Complexity of language

A Lisa specification defines the instruction set of a processor as well as its microarchitecture. To describe different microarchitectural aspects, a variety of concepts and language constructs are defined in Lisa. Just to give an impression: Lisa defines over 120 Keywords. The language may therefore be considered rather complex and hard to learn. In addition, a user likely needs knowledge on microarchitectures, programming in C and VHDL.

ViDL in contrast features a lean design with a clear set of very powerful and orthogonal concepts. The user neither needs knowledge on processor microarchitectures, nor on other languages, such as C or VHDL.

3.6.6 Practical application

In the following, the results of four recent publications on the application of Lisa are presented.

Meyer-Bäse et al. [38] describe how Lisa may be applied in teaching students, using the example of two academic processors, namely URISC and ERISC. The URISC processor is a modified "one instruction set computer", which is very similar to the OISC that has been defined in ViDL. The ERISC implements a rather small set of 13 basic instructions, which includes arithmetic and logic instructions. Aspects, like conditional execution, 64-bit multiplications or delayed branches are not mentioned.

Both processor specifications use a two-stage pipeline, which is structurally equivalent to a non-pipelined processor. Typical aspects of pipelines, such as forwarding, interlocking and speculative execution are not applicable for this microarchitecture. A faster implementation of the instruction sets requires a longer pipeline and the aforementioned techniques for hazard-resolution. Both has to be specified in Lisa by the developer. The simplicity and maintainability of the specifications can not be judged, as they have not been published. Basic figures on the length of the specification or the required development time are not mentioned. The applicability of Lisa in education can therefore not be estimated on the foundation of this paper. The authors do not state, that students have specified the the URISC and ERIS processors in Lisa. Experiences from teaching and applying Lisa are not mentioned.

Jae-Jin Lee et al. [36] propose an application specific processor, which is implemented using Lisa. The instruction set consists of a set of basic instructions and 9 application specific instructions. The total number of instructions and their semantics are not described. The resulting processor can therefore not be compared to other processors. The authors state, that the Lisa specification

	VHDL	Lisa	ViDL
Length of specification	291 SLOC	1176 SLOC	207 SLOC (97 LLOC)
Development time	1 week	1 month	90 minutes
FPGA Clock Speed	61.1 MHz	44.3 MHz	
FPGA Slices	650	3489	

Table 3.1: Comparison of VHDL, Lisa and ViDL. Data for VHDL and Lisa has been reported by Franz [16].

uses a 6-stage pipeline, which is fully bypassed. Further information on how it is implementation in Lisa is not given. The complexity of bypassing, interlocking and branch prediction in Lisa can therefore not be estimated. The specification seems not to be published. The dynamic behavior of the processor by means of branch penalties and instruction latencies is not described. Characteristics of the Lisa specification, such as length, maintainability and development effort are not given. Synthesis for an 180 nm TSMC standard cell technology yields a clock frequency of 100 MHz. Estimates on the area requirement and power consumption are missing.

Dodani et al. [8] present an implementation of a simple processor in Lisa. The processor features a usual set of 19 basic instructions. Exceptional instructions or complex addressing modes are not mentioned. The Lisa specification uses a 3-stage pipeline. Resolution of hazards by means of forwarding, interlocking and speculative execution is not mentioned. The authors state, that the processor has been synthesized for a 65 nm TSMC standard cell technology, resulting in an estimated chip area of $0.075\,\mathrm{mm}^2$. It is not stated, which components (datapath, register-file, memory, multiplier) have been included in the synthesis. An estimated clock frequency is not mentioned. The Lisa specification or figures on its length and development effort are not given.

Franz [16] has evaluated Lisa in his master thesis. He specified two processors in Lisa, one of which is the *simple reduced instruction set computer* (SRC), which is defined in the book *Computer Systems Design and Architecture* [25]. The instruction set consists of 28 instructions. Franz's supervisor implemented the SRC instruction set in VHDL and Franz in Lisa. Franz characterized his supervisor as an experience VHDL developer and himself as a new Lisa developer. After specifying a non-pipelined version of the SRC processor in Lisa (which took two month), he defined another pipelined version in Lisa. For the latter, the reported development time and length of specification is shown in Table 3.1. The table also shows the time it took me to specify the SRC instruction set in ViDL. For the ViDL specification, simulation on an FPGA has been left out, as it requires the same prototyping hardware and setup for a fair comparison. However, a generated 5-stage processor yields a clock frequency of 770 MHz on a 65 nm low power technology. The ViDL specification of SRC has been published [10], to allow for a direct comparison with the Lisa specification that Franz has published ([16] Appendix B; pages 109–137).

Franz states, that Lisa has a steep learning curve and reported a number of

pitfalls, which he encountered during development. Examples include undocumented aspects, unsupported features and modeling techniques that do not work or yield poor code. He states, that "Pipeline stalls proved to be a hard feature to model with the LISA language.". He mentions, that it took quite long to debug pipeline control and fix the specification.

To sum up, most reports on Lisa give only few facts and figures on the actual specification and the development effort. The resolution of data-hazards and control-hazards is hardly discussed, although this is a key aspect of pipelined processor implementations. Franz has published a detailed report on the use of Lisa, which also reveals weak points.

3.7 ISDL

The *Instruction Set Description Language (ISDL)* is targeted at the generation of compiler tools. Necessary language aspects for the generation of an assembler are discussed [22]. The generation of a compiler is announced, but no literature is available on this topic. A technical report on ISDL was published, but is no longer available. Some aspects of ISDL can therefore not properly be discussed in the following. Due to the lack of information, ISDL can not be assigned to one of the language classes that have been defined in Section 3.1.

Storages As in other processor description languages, storages are modeled as arrays. Each array has a specific width and element type. The element type is a bit-string of a specific width. The special case of a single register is modeled as a scalar, which contains a bit-string of a specific width.

Instruction Set Structure Similar to nML, the set of instructions is intensionally defined by a context free grammar $G = (T, N, P, S)$ in ISDL. The grammar seems to be derived from the syntactic structure of assembly instructions. This is questionable, as the syntactic structure of assembly instructions may not match the logical structure of an instruction set.

As in nML, an instruction corresponds to a derivation tree of a grammar G. The set of all instructions therefore corresponds to the set of derivation trees. In addition to nML, rules may be specified in ISDL, which further restrict the considered set of derivation trees. This restricted set is called the set of "valid instructions". A derivation tree of this set obeys all specified rules. Practically, rules are used to express constraints between instructions in a VLIW.

Technically, a rule is a Boolean expression on the symbols of G. A derivation tree implies an interpretation of all symbols: The interpretation of a symbols $X \in N \cup T$ is *true* iff the derivation tree contains a node of X. To belong to the set of valid derivation trees, the interpretation of all rules must be *true*.

Instruction Semantics Instruction semantics are constituted from the semantics of non-terminals, as in nML. The semantics of non-terminals are specified in

an undefined language, which reminds of C. A formal definition of the language is not given.

Summary ISDL is in great parts similar to nML. In addition, a set of rules can be defined to restrict the set of instructions. The instruction semantics seem to be defined in a C-like language, which is not well suited to generate processor implementations in terms of HDL code.

3.8 Expression

The specification language *Expression* [23, 39, 21] belongs to the class of processor description languages (PDL), as it describes a processor on a microarchitectural level. Roughly speaking, a set of resources and their interconnection is defined in Expression. In particular, a pipeline is defined as a sequence of stages. Each stage may be composed from multiple parallel units. Each unit may again be a pipeline or an execution unit. An execution unit is associated with a set of instructions. An instruction is defined by its encoding, its assembly notation and its semantics.

Literature gives only an impression of the specification language. Questions on soundness and completeness arise, but are not addressed. For instance: May two pipelines of different length be arranged as parallel units and what does that mean for the overall pipeline? What language is used to specify instruction semantics? Examples in [23, 39] apply only basic arithmetic operations (+ - *). Sophisticated examples, that would demonstrate the expressiveness (like "count-leading-zeros") are not given. In an early technical report [21], the context free grammar of Expression does not define the structure of the semantics specification. The corresponding non-terminal is only annotated with the comment "`% the semantics of the operation`". A related point that is not covered is the width of operations.

3.9 Tensilica instruction extension (TIE)

As the name suggests, the Tensilica Instruction Extension (TIE) language [19, 53, 50, 37] is intended to define instruction set extensions. An existing instruction set can be extended by application specific instructions to accelerate hot-spots of applications. In contrast to related approaches, the basic instruction set (Xtensa core) is fixed and remains unchanged. TIE primarily targets the scenario of instruction set extension. The specification of complete ISAs seems not to be intended and respective examples are not mentioned in literature.

TIE is based on a subset of the hardware description language Verilog [47]. Operations and the type system of TIE are for instance inherited from Verilog. Among all specification languages, TIE is most similar to ViDL. Both languages share common aspects, such as the distinct specification of instructions and a

certain abstraction from processor implementations. Nevertheless, there are still significant differences, which are discussed later.

TIE can mostly be regarded as an instruction set description language, according to the taxonomy in Section 3.1. However, the abstraction from microarchitectural aspects is not as strict as in ViDL. Two exceptions are discussed later in Section 3.9.3 and 3.9.4.

3.9.1 State

The ISA state can be extended by user defined registers and register files. The width of a register or register file must lie in the range of 1 to 1024 bits. The depth of register files has to be a power of two up to 1024. These limits are acceptable for realistic instruction sets. The instantiation of additional data memories is not mentioned in literature [50].

3.9.2 Instruction semantics

The semantics of an instruction are defined by a sequence of assignments, which are "executed" concurrently. To support conditional assignments, a write disable expression (negation of a write enable expression) can be defined for each register.

```
reg      = <valueExpr>
reg_kill = <writeDisableExpr>
```

The value `<valueExpr>` is only written to the register `reg`, if `<writeDisableExpr>` evaluates to *false*. Using this concept, conditional branches and conditional execution can be specified. The concept of a write disable expression in TIE is a special case of epsilon logic in ViDL, which is discussed in Section 4.6.

Operations To define semantics of an instruction, TIE provides about 40 primitive operations. These are mainly inherited from the hardware description language Verilog. The set of operations in TIE corresponds to the set of transfer primitives in ViDL. In contrast to TIE, the set of ViDL's primitives can be extended by the user, as described in Chapter 5.

Wires To reuse the value of an expression, TIE inherits the concept of *wires* from Verilog. The value of a wire is defined by assigning an expression. The value can then be referenced by the name of the wire. The concept is similar to variables in ViDL. The width of a wire is specified explicitly by the user. In contrast, the width of a variable in ViDL is derived by the generator (Section 4.9.2) using type inference.

Functions To encapsulate common semantics, TIE features the concept of functions. A TIE function has an arbitrary number of parameters and yields exactly one result. Parameter and result types are specified explicitly along with the function definition. In addition to TIE, ViDL enables the specification of

- polymorphic functions,
- higher order functions (functionals) and
- functions with multiple results

The advantages of these concepts are discussed in Section 4.5. Basically, the degree of reuse is increased and the definition of computation patters (e.g. SIMD) is enabled.

Bit-widths To interpret an expression in TIE, the bit-width of each application of an operation needs to be known. TIE defines these widths by an algorithm that is inherited from Verilog and is defined in the IEEE 1364 standard [47]. Given an expression tree, the algorithm computes the bit-width of each application of an operation in the tree. To understand a Verilog resp. TIE specification, a developer needs to know this algorithm as well as width-computations that are specific to each operation. This knowledge is for instance essential to understand the semantics of the expressions `(a+b)>>1` and `(a+b+0)>>1`. Although the expressions seem to be semantically equivalent, they may yield different results, according to bit-width inference. In contrast to Verilog and TIE, ViDL defines bit-widths by means of sub-typing (Section 4.9) rather than an algorithm. As a result, bit-width rules of operations can concisely be expressed by signatures in ViDL.

Operands An instruction has a number of explicit and implicit operands. An operand may be either an immediate or a register operand. For register operands, the respective register file and the direction (in, out or inout) are specified. Memory however is treated differently from registers. It is regarded as an interface, which consists of 5 signals: address, data-in, data-out, read-disable and write-disable. The latter two signals allow byte-precise disabling (enabling) of read and write accesses on the memory. This distinct concept for the data memory is related to the write disable expressions for registers. However, both concepts are defined separately in TIE. In ViDL, these concepts are subsumed by epsilon logic. As a result, a memory can be defined and used as simple as a register file in ViDL. There is no need for a special memory interface, as in TIE.

3.9.3 Hardware sharing

Hardware sharing between instructions is defined explicitly in TIE. The semantics of several instructions is defined in a so called `semantic` block. To distinguish between instructions, a Boolean variable X is introduced for each instruction X. The variable X is true, if and only if the respective instruction is executed. ViDL does not include such constructs, as hardware sharing is not considered part of an instruction set.

3.9.4 Datapath scheduling

The `schedule` directive in TIE is used to explicitly specify the relation between datapath and pipeline. In particular, the publication stage of an instruction result is specified. Technically, this is the stage in which a result is fed into the bypass. Effectively, the `schedule` directive thereby defines the bypass and the resulting interlocking. In addition, the `schedule` directive is used to assign the stages of read ports and assign intermediate signals to stages.

Scheduling aspects are part of the microarchitectural implementation and therefore not defined in ViDL. Instead, these aspects are derived automatically by the port scheduler (Section 7.6.4) and forwarding generator (Section 7.6.7), which are part of the processor generator.

3.10 DPG — Datapath generator

The datapath generator (DPG) presented by Weiss et al. [55] is actually not directly related to the domain of processor and ISA specification. However, it has been considered, since ViDL's processor generator also produces a datapath. However, ViDL operates on RTL level, whereas the DPG focuses on the physical level. In particular, the generator instantiates parameterized hard macros[3]. The input is a signal flow graph (SFG) which is denoted in the domain specific language DDL. It includes the definition of delays and is related to the register transfer level.

Compared to ViDL, the DPG helps to implement a specific signal flow on a low level of hardware design. It focuses on the implementation of algorithmic kernels rather than on the implementation of instruction sets. This is a very different domain. The DPG does for instance not synthesize processor structures like decoder, register ports or a complete pipeline including bypass and forwarding. Therefore, the DPG gives the developer control over the physical implementation using parameterized macros. ViDL on the other hand postpones the physical implementation to RTL compilers, which map the RTL description to a standard-cell technology.

[3]Parameterized logical primitive with an associated physical design for a certain chip technology

Chapter 4

ViDL — Versatile ISA description language

A language designer typically defines a language with certain design goals in mind. During the development of ViDL, three major goals have been considered. First, ViDL should support typical *development processes* of instruction sets. Typical processes and design scenarios have been introduced in Section 2.2. In particular, the scenarios of rapid design space exploration (DSE) and instruction set extension (ISE) are of interest. Second, the language should have a high quality with respect to general *language design criteria*, which have been outline in Section 2.3. A language that obeys these criteria is expected to reduce development time, avoid specification flaws and increase maintainability. Finally, the language should be *powerful*, to specify serious real-world instruction sets in a natural way. A language that covers only artificial instruction sets that are tailored to the language will certainly not be accepted by developers from industry. ViDL was developed against the background of existing instruction sets, such as ARM, MIPS, Power and CoreVA (Section 2.1). Each of these instruction sets challenges ViDL in a different way, as each instruction set uses a dedicated set of concepts. For instance, ARM features conditional execution, branches of MIPS have delay-slots and CoreVA defines a set of condition registers. In the following, ViDL's design goals are discussed along with each language concept.

To give a basic impression of ViDL, a small specification is presented in the next section. Typical specifications are clear and reliable, as ViDL is a highly structured language (Section 4.2). Besides, it strictly abstracts from microarchitectural aspects (Section 4.3), which simplifies specification and allows to generate very different products. Similar to instruction set manuals, instructions are defined separately in ViDL (Section 4.4). Nevertheless, common behavior can be factored out using concepts from functional programming languages (Section 4.5). Further concepts have been developed, which are specific to the domain of instruction set specification. For instance, conditional and partial write accesses of instructions can concisely be defined using epsilon logic (Sec-

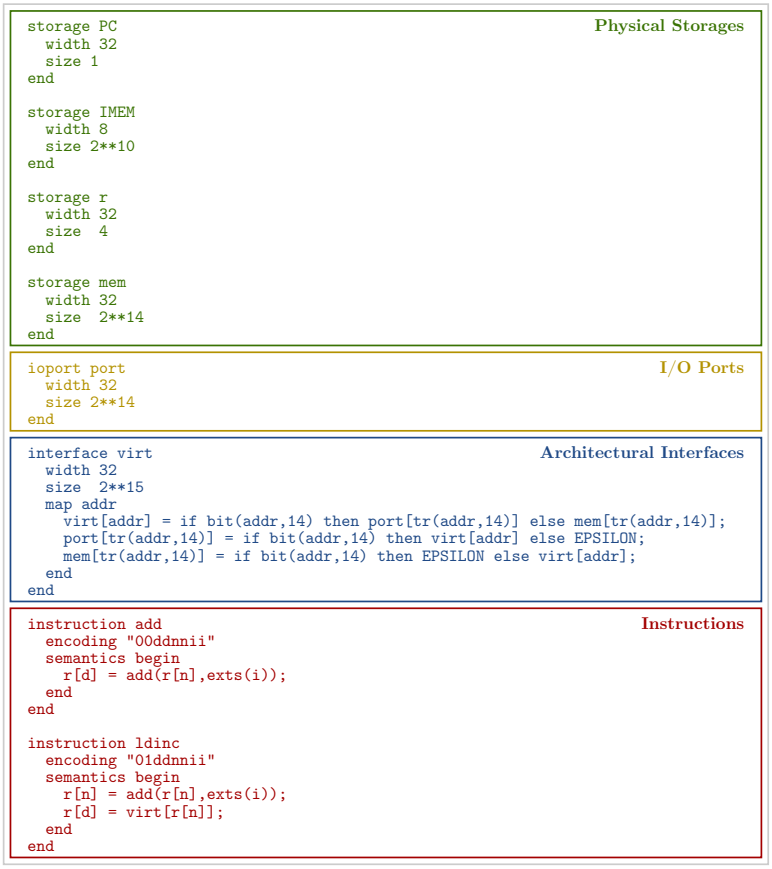


Figure 4.1: Example of a simple and complete ViDL specification.

tion 4.6). The temporal behavior an instruction (e.g. delay-slots of branches) can directly be denoted as a so called delay (Section 4.7). Using architectural interfaces (Section 4.8), views on storages and I/O ports can be defined, which greatly improves the quality of instruction set specifications. Finally, instructions can be specified without denoting the bit-width of operations, as ViDL is implicitly typed (Section 4.9).

4.1 A ViDL example

To give a rough impression of ViDL, Figure 4.1 shows a simple, yet complete ISA specification. The example includes four storages, one I/O port, an architectural interface and two instructions. It is only meant to give a rough impression of ViDL, and does not demonstrate all concepts and capabilities of the language.

The storages `PC` and `IMEM` are mandatory and define the program counter and

the instruction memory. The program counter is modeled as a normal storage[1], since it is part of the processor state.

A register file `r` is specified to consist of 4 registers, each 32-bit wide. In the same way, a data memory (`mem`) is specified to contain 2^{14} 32-bit words. These four storages constitute the state of the instruction set. For I/O, one port (`port`) is defined similar to the memory. The port has a data-bus of 32-bits and an address space of 2^{14} words. The next declaration defines an architectural interface (`virt`), which models a virtual address space. The memory is mapped into the lower and the port into the upper half of the space. In other words, the architectural interface implements memory mapped I/O. Further applications of architectural interfaces are described in Chapter 6.

The specified ISA defines two instructions `add` and `ldinc`. Each instruction is defined independently by its encoding and its semantics (behavior). For the encoding, a notation similar to ISA manuals is used. The `add` instruction is encoded in 8 bits, where the two most significant bits are zero. The remaining bits encode three operands named `d`, `n` and `i`. These names are then used in the instruction's semantics section to refer to the encoded operands. Note that the developer does not have to distinguish between register operands and signed/unsigned immediates. The addressing mode is instead specified as part of the semantics.

The semantics of an instruction are defined by a set of assignments. The `add` instruction contains one assignment to the destination register `r[d]`. The right hand side of the assignment defines the value to be assigned. For `add`, this is the sum of register `r[n]` and the sign extended immediate `i`.

The `ldinc` instruction loads a word from the data memory into the destination register `r[d]`. It uses the addressing mode "Post-Increment", as defined in [24], i.e. the memory is addressed by register `r[n]`, which is then incremented by the signed immediate `i`. For each of the two operations (load and increment), one assignment is specified. The assignments are executed concurrently, i.e. the assignment to `r[n]` does not have an effect on the expression `mem[r[n]]`. Hence, assignments in ViDL behave like signal assignments in VHDL. The set of all instructions constitutes the transfer function of the instruction set.

Both instructions contain the common subexpression `add(r[n],exts(i))`. A hardware related reader may be inclined to factor this common aspect out, to reduce hardware complexity. Fortunately, this is done automatically by the processor generator, so the user can focus on instruction set development. Actually, optimization driven factorization is even counter productive. It introduces logical dependences between otherwise unrelated instructions, which corrupts maintainability.

Note, that aspects of the microarchitecture are not specified in ViDL, as ViDL focuses on the instruction set. Instead, microarchitectural aspects are derived automatically by the processor generator (Section 7.6). This includes the pipeline structure, forwarding, interlocking, speculative execution and register ports.

[1] Nevertheless, the generated implementation significantly differs from normal registers.

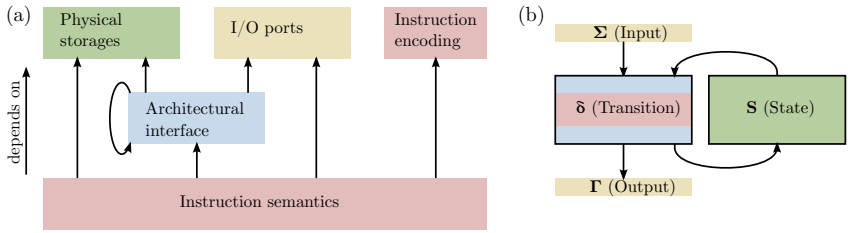


Figure 4.2: Structure of an ISA specification in ViDL.

4.2 Structure of a specification

On an abstract level, an instruction set can be regarded as a finite-state machine. It has a limited amount of registers and memory (state), an interface for communication (input and output) and a set of instructions, which define the behavior (transition).

This simple structure is the foundation of ViDL. The components of a ViDL specification and their relation are shown in Figure 4.2a. It defines a set of *physical storages*, which constitute the state of an instruction set. A physical storage may for instance be a register file or a data memory. The communication interface of an instruction set is defined by a set of *I/O ports*. The behavior (transition) is defined by the set of all instructions. An instruction itself is defined by its *encoding* and its *semantics* (Section 4.4). The specification of semantics refers to storages, ports and the instruction encoding. It typically accounts for the greatest part of an ISA specification. In addition, ViDL features so called *architectural interfaces* (Section 4.8). An architectural interface basically models a view on storages and ports. It is an optional layer of abstraction, which leads to a clearer ISA specification and increases its maintainability.

As Figure 4.2a shows, the components of a ViDL specification are loosely coupled. The instruction's semantics can for instance be changed without touching the specification of physical storages. The same holds for storages, ports and the encoding, which are completely decoupled. Instructions can be added and removed without reconsidering other parts of the specification. This feature is important in the context of instruction set extension and design space exploration.

Figure 4.2b illustrates ViDL's execution semantics in terms of a finite-state machine. The interface of the finite-state machine (Σ and Γ) is defined by the ISA's I/O ports and the state S by the set of physical storages. The transition function is defined by the set of all instructions and architectural interfaces.

4.3 Abstraction from microarchitecture

A ViDL specification strictly focuses on aspects of the instruction set. Aspects of the implementation (microarchitecture) are not defined, for the sake of abstrac-

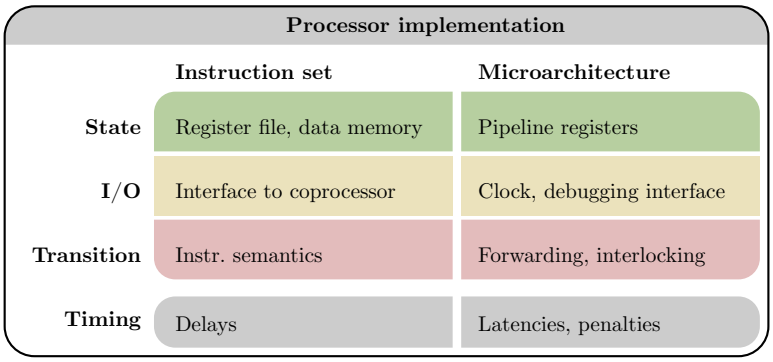


Figure 4.3: Aspects of instruction sets and micorarchitectures.

tion and simplicity. Instead, all implementation specific aspects are contributed by the generators. Figure 4.3 shows examples of instruction set related aspects and microarchitectural aspects of a pipelined processor.

State This thesis divides the state of a processor into an ISA part and a microarchitectural part. The ISA state contains all storages that are exposed to a program, such as general purpose registers and data memories. The microarchitectural state includes additional storages of the microarchitecture, such as pipeline registers. The microarchitectural state is only introduced to implement the microarchitecture and is thereby implementation dependent. It is not exposed to the application, and can not be accessed by instructions.

I/O The I/O interface of a processor consists of instruction set specific interfaces, and interfaces that are introduces by the microarchitectural implementation. An example of the former is an interface to a coprocessor, which is typically accessible by dedicated coprocessor instructions. Examples of the latter include the processors clock signal and a debugging interface (e.g. JTAG).

Transition The behavior of the processor is constituted from the instruction's semantics and microarchitectural control. The latter includes branch prediction and interlocking, which is entirely contributed by the processor generator. ViDL strictly abstracts from such pipeline control.

Timing This thesis distinguished two kinds of timing. Timing that affects program semantics and timing that affects execution speed only. Delays of instructions (e.g. delay slots of branches) affect data-flow or control-flow in a program and are therefore a part of the instruction set. Latencies and branch penalties in contrast are the result of hazard-resolution and do not affect program semantics. They are implementation defined and need not be considered

to interpret a program. However, latencies and penalties have an impact on execution speed, in terms of CPI[2].

4.4 Instructions

The definition of instructions accounts for the greatest part of a ViDL specification. An instruction is defined by its encoding and its semantics by means of register transfer.

4.4.1 Encoding

The encoding of an instruction is defined by a pattern. The pattern consists of literal bits (0 and 1) and fields. The literal bits constitute the opcode of the instruction, i.e. they are used to identify an instruction. Fields are used to encode operands such as register numbers and immediates. Each field is associated with a one letter identifier. This identifier can be used in the instruction semantics to refer to the field's value. Note that the purpose or sign extension of a field is not an aspect of the encoding. These aspects are therefore specified as part of instruction semantics.

The pattern `01dd10iii` for example defines an 8-bit instruction with two fields d and i. The example demonstrates that the opcode (`0110`) may not be encoded in adjacent bits of the instruction. This is an important feature, which is required to specify most realistic instruction sets. The notation in ViDL is very similar to that in instruction set manuals. Formalization of existing instruction sets is thereby simplified. In contrast, other approaches require the user to manually analyze the instruction space and define a decision tree. ViDL shifts this analysis from the developer to the generator. The generator also checks for conflicting instruction encodings. Overlapping instructions are identified and reported to the user. Hence, copy-paste errors and typing errors are typically detected. Although the encoding is defined in a declarative manner by a set of patterns, the generated decoders are very efficient.

4.4.2 Semantics

The semantics of an instruction are defined by a set of assignments. Each assignment sets a storage, port or architectural interface to a given value. The assignments are "executed" concurrently, similar to signal assignments in VHDL. The following example defines a simple instruction that swaps the contents of registers `r[a]` and `r[b]`.

```
instruction rswap
  encoding "0101aabb"
  semantics begin
    r[a]=r[b]
```

[2] cycles per instruction

```
    r[b]=r[a]
  end
end
```

Note that the assignments are executed concurrently. If this would not be true, the swap instruction could not be defined in such a concise way.

As a result of concurrent execution, assignments do not interfere. Unintended side-effects are excluded, which increases reliability of the ISA specification. Each assignment can be regarded without considering other assignments of the instruction. Besides, the order of assignments does not affect instruction semantics, as it is typical for declarative languages.

The set of assignments can be regarded as a definition of the next processor state. Storages on the right hand side of assignments refer to the current processor state and storages on the left hand side refer to the next processor state. The referred state can be specified explicitly, using the concept of "delays" (Section 4.7).

4.5 Functional concepts

ViDL belongs to the class of functional languages, which is a subclass of declarative languages. The semantics of instructions are specified in a dialect of SML (Standard Meta Language). A functional language has been chosen as the foundation of ViDL, since it allows for clear specification of instructions. Semantics can be defined in a declarative manner, free of unintended side effects. In contrast, an imperative language is based on the principle of side effects. Besides, functional languages are a super set of dataflow languages. Data flow languages are by their nature well suited to describe register transfer of instructions. The same holds for functional languages, as long as recursion is excluded as in ViDL. In addition to a classical dataflow language, ViDL inherits sophisticated concepts, such as polymorphism and higher order functions. Both enable a significant degree of reuse, which is crucial for concise specifications of instruction sets.

In particular, ViDL includes most concepts of SML, such as variables, conditions (`if`), switches (`switch`), tuples, patterns, lambda expressions, higher order functions, polymorphism and type inference. In addition, ViDL defines further concepts that are specific to the domain of ISA specification, such as vectors and bit-string literals. Like SML, ViDL is statically and implicitly typed using type inference. The type system however is very different, as instructions operate on bit-strings rather than integers and floating point numbers. ViDL defines two polymorphic types, to model the width of bit-strings, among other types. The type system and the respective type inference are described by Dreesen et al. [14] and in Section 4.9.

4.5.1 Functions

Functions in ViDL are similar to functions in SML. A named function can be defined using the notation `fun f (...) = ...` in the context of a definition. An anonymous function can be defined by a lambda expression `fn (...) => ...` in the context of an expression. As in SML, the parameter and result of a ViDL function is a tuple, which enables the definition of functions with multiple parameters and multiple results (Section 4.5.6).

ViDL supports higher order functions (functionals), which means that functions can be passed as arguments and returned by functions. The following functional `vect2` applies a function `f` on the components of two dimensional vectors. The function and both vectors are passed as arguments to `vect2`.

```
fun vect2(f,(ah,al),(bh,bl)) = ( f(ah,bh) , f(al,bl) )
```

4.5.2 Polymorphism

Functions are polymorphic with respect to bit-widths. The following function concatenates three bit-strings. It can be applied for any widths x, y, z of `a`,`b` and `c`.

```
fun cat3(a,b,c) = cat(cat(a,b),c)
```

Its signature is $E_x \times E_y \times E_z \mapsto E_{x+y+z}$, where x, y, z are integral type parameters. The type system is discussed in Section 4.9. Due to polymorphic functions, dataflow of arbitrary width can be specified in ViDL. The dataflow is not limited or restricted to certain bit-widths. The specification of 4711-bit arithmetic is as simple as the specification of 32-bit arithmetic.

4.5.3 Closures

In ViDL, functions may use variables from their definition context. These variables are added to the function's closure. A function may for instance refer to a register or to a field of an instruction. Closures are properly implemented in ViDL. In the following example, the application of `f` yields the value 1, as expected. The use of `x` is bound to the first definition of `x`, which is valid in `f`'s definition context.

```
let
  x = 1,
  fun f() = x,
  x = 2
in
  ... f() ...
end
```

```
r[d] = let
         sum  = add(r[a],r[b]),
         overflow = bit(sum,32)
       in
         if overflow then 0xffffffff else sum
       end
```

Figure 4.4: Scopes and bindings introduced by an application of let.

4.5.4 Recursion

As a major difference to SML, functions must be non-recursive in ViDL. This property is checked statically by the generator. ViDL imposes this restriction, to ensure soundness and to enable the generation of efficient hardware and software implementations. Actually, recursion is not needed to specify realistic instruction sets. As a consequence, recursive data-types and lists are not integrated in ViDL as well. There is no meaningful application of a recursive data-type in the absence of recursive functions. As ViDL does neither include loops nor recursion, the simulation of instructions is guaranteed to terminate. In general termination is undecidable.

4.5.5 Name binding

In functional languages, name bindings denote the association of expressions and identifiers. For example, a definition like `v=add(47,11)` binds the expression `add(47,11)` to the identifier `v`. In the scope of this definition, the variable `v` represents the expression `add(47,11)`. Any use of `v` is conceptually substituted by the bound expression. A binding is different from an assignment, in the way that a binding can not be "changed". ViDL has 3 constructs to defined bindings: `let` expressions, `semantics` blocks and `define` sections. There are further sources of bindings in ViDL, which are not directly exposed to the developer and therefore not described in the following.

The let construct in ViDL is defined just as in SML, including scope rules. In practice, `let` is used to introduce local bindings within expressions. A frequent subexpression can be bound to a variable, which is then used in its place. The example in Figure 4.4 defines a saturating add instruction, using two bindings, where the first binding `sum` is used twice. The scopes of both bindings are also indicated by blue and yellow highlighting. Note that the first binding is valid in the definition of the second binding and the scope of the second binding is a subscope of the first one. This small example already gives a rough impression on how bindings can improve an ISA specification. Subexpressions can be given meaningful names, such as `sum` and `overflow`, which improves readability. Common subexpressions can be factored out, which increases reuse and thereby consistency and maintainability.

The `semantics` block is very similar to the let construct, but its scope ranges over an entire instruction, i.e. over all assignments. This scope can not be covered by `let`, since `let` is an expression and thereby limited to a single assign-

ment. The `semantics` block is typically used to define operands, and expressions that are used in different assignments.

The `define` block has the largest scope. It reaches from the definition to the end of specification. It is typically used to define common behavior of instructions. A typical example is the definition of an addressing mode, which is used by many instructions. Note that all definitions in the `define` block are instruction independent by design. In contrast, definitions in the `semantics` block are likely instruction specific and can use fields of the instruction encoding.

4.5.6 Tuples

ViDL implements the concept of tuples. As in SML, the parameter and result of a function is regarded to be a tuple. This enables functions with multiple parameters and multiple results. Hence, an HDL entity with multiple in-ports and out-ports can be resembled in ViDL by a function with multiple parameters and results. The components of a tuple can be accessed by binding their values to dedicated variables, using a pattern. In the following code, the `addx` function is assumed to return a pair. The components of that pair are bound to variables `sum` and `carry`

```
let (sum,carry) = addx(x,y) in ... end
```

If the structure of the left and right hand side of a definition is not compatible, the generator reports an error, including the position of the definition. Tuples may be nested arbitrary, which enables more sophisticated data structures. A vector of three complex numbers may for instance be represented by a tuple `((r1,i1),(r2,i2),(r3,i3))`.

Tuples can be used to specify SIMD and Vector instructions. A vector of size n is simply represented as an n-tuple. Components can easily be accessed and the specification becomes more readable. Tuples also offer a simple way to define and use functions with multiple (related) results. For instance, the `split` function divides a value into its high and low part. An ISA specific `evalFlags` function may return the 4 status flags (Z,N,C,V) for a given value. An ISA specific `addrMode` function for pre/post-increment load/store instructions may return the effective address and the new value for the base register.

4.5.7 Vectors

ViDL uses a common notation to refer to storages, I/O ports and architectural interfaces. All three concepts use the notation of vectors. For instance, the vector `foo[x]` may refer to a storage, port or interface called “foo”, where `x` denotes the address.

The common notation not only simplifies the language, but also increases its maintainability. Storages and ports and architectural interfaces can easily be exchanged, without touching the specification of instructions. A physical storage `mem` may for instance be replaced by an architectural interface `mem`, which models a virtual address space, as initially shown in Figure 4.1. Such

rapid and reliable modifications of the ISA are especially important in the DSE scenario. Again, instructions remain unchanged by these modifications.

4.5.8 Review of concepts

In the following, the functional concepts that have been presented so far are briefly reviewed with respect to general language design criteria. The concepts are intentionally reviewed at this point, to keep the discussion close to their explanation. Design criteria of domain specific languages have been introduced in Section 2.3.

Reuse

The concepts described so far allow for different kinds of reuse. The simplest form is the reuse of an expression. An intermediate result can be bound to a variable, which is then used multiple times. A higher degree of reuse is provided by functions. A function abstracts from expressions and can be used to define recurring behavior such as an addressing mode. An addressing mode function is instruction independent and can therefore be used throughout the whole ISA specification. As functions in ViDL are polymorphic, they also abstract from bit-widths. A function can therefore be applied in contexts of different widths.

The highest degree of abstraction and reuse in ViDL is offered by functionals. Functionals do not only abstract from expressions, but also from behavior, which is contributed by the "caller" as an argument. A functional can be used to define a pattern for defining a series of similar instructions. The instruction specific behavior is encapsulated in a function and passed to the functional. The functional only defines the semantics that are common to all instructions.

A functional can for instance be used to define the SIMD concepts, as shown in the following.

```
fun simd(f,(a1,a2),(b1,b2)) = (f(a1,b1),f(a2,b2))
```

The `simd` functional leaves the actual operation on vector components open. The operation is supplied by the application context via the parameter `f`. The applications of `f` can be regarded as extension points.

Most functions and functionals are ISA independent and can therefore be added to libraries. These libraries may then be reused among different instruction set specifications. ViDL defines a standard library, which includes many practical functions and functionals for ISA specification. The library includes typical addressing modes, functions for condition evaluation and functionals for SIMD instructions. Besides, the library defines convenience functions, which simplify ISA specification. The `zeros(n)` function for instance yields a sequence of n zero-bits and is defined as

```
fun zeros(n) = trunc(0,n)
```

Note that `zeros(n)` is neither defined as part of the language, nor as a primitive, but as a library function within ViDL. Such functions can be added by the ViDL user without touching the generator.

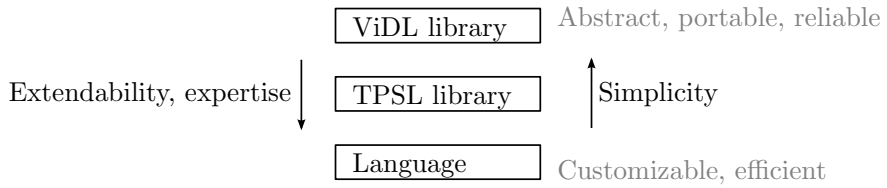


Figure 4.5: Extending ViDL on different levels.

Extendability

The functionality of ViDL can be extended on different levels, as shown in Figure 4.5. The easiest ways to extend ViDL is to define a function or functional in the ViDL library. The developer only needs to know ViDL to implement such extensions.

If some functionality can not efficiently be defined by combining existing functions, a primitive may be defined in the so called TPSL library. TPSL is used to define transfer primitives, such as `and` or `not`. It is described in detail in Chapter 5. The definition of a transfer primitive is more complex, as code generators have to be defined for each target language (e.g. C and VHDL). To perform such an extension, the user needs to known ViDL, TPSL and the target languages.

Extending the language by new constructs offers most flexibility, but is also most laborious. The grammar of ViDL and the generators need to be adapted, which involves excessive testing. The user requires knowledge on good language design and compiler construction. The extension must be sound and should not interfere with existing language concepts. As language extensions are costly, the ViDL library and the TPSL library are provided as primary extension points. Fortunately, even concepts like SIMD can be defined as functionals in the ViDL library and do not require dedicated language supports.

Simplicity

The existence of closures in ViDL simplifies specification. A developer is not forced, to pass values such as instruction operands via parameters. Instead, functions can use variables from their definition context. In the following example, the function `bias` uses the immediate `i`, which is defined by the instructions encoding.

```
encoding "1100iiii"
semantics
  fun bias(b) = add(zero(i),b)
begin
  ... bias(-64) ... bias(-128) ...
end
```

Variables and functions not only offer a way of reuse, but also help to increase readability. Complex expressions can be broken down to smaller pieces, which are assigned descriptive names. In the following example, both semantics blocks describe the same behavior and result in the same implementation. The second block however is clearer, as subexpressions are bound to descriptive variables.

```
semantics begin
  pc[0] = if ne(sub(r[a],r[b]),0)
            then add(pc[0],ext(i)) else EPSILON
end

semantics
  dist   = exts(cat(i,0b00)),
  target = add(pc[0],dist),
  cond   = ne(sub(r[a],r[b]),0)
begin
  pc[0] = if cond then target else EPSILON
end
```

Reliability

Errors and flaws in software and hardware are frequently introduced by unintended side effects. In ViDL, the effects of an instruction are only defined in the semantics block in terms of assignments. To understand the effects of an instruction, the developer need only regard this block and can be assured, that the instruction does not have any further side effects on the processor state. As the effects of instructions are clear, unintended or unconsidered side effects are excluded. A classical source of flaws is therefore closed.

Another major source of flaws is the resolution of hazards by means of pipeline control. Such errors likely show up, for exceptional combinations of instructions under certain circumstances. Such exotic corner cases may not be covered during testing and the flaws are therefore not detected. In ViDL, such flaws are excluded by *generating pipeline control* and by *decoupled instructions*. Pipeline control is entirely generated, by generic construction methods. The developer is not responsible to specify pipeline control that correctly considers all feasible instruction streams for all possible pipeline states. This not only simplifies the specification, but also excludes a major source of processor flaws.

Second, flaws are excluded by the fact that instructions are completely decoupled in ViDL. Instructions only interact via the ISA state, which is clearly defined by the ISA storages. The semantics of an instruction are not affected by side effects of other instructions. As a result, a ViDL developer does not have to reason about the execution context of an instruction.

Learning Curve

ViDL should offer easy access for new developers. It is based on the functional programming language SML and should therefore be familiar to developers, who

have a basic understanding of functional programming. Developers without such background need only learn a small set of basic concepts. Besides, only few of these concepts need to be considered, to specify basic instruction sets. Concepts, such as I/O ports, architectural interfaces, functions, and tuples can be ignored at the beginning. They can be learned after first steps with ViDL have succeeded.

Thanks to abstraction, a developer only needs to known ViDL and the principles of instruction sets. Expertise on microarchitectures, simulator implementation, C and VHDL is not required. For instance, a developer does not have to know, how a pipeline bypass is implemented and how pipeline hazards are correctly resolved.

Compared to other ISA and processor specification languages, ViDL uses a simple model for instruction sets. It consists of storages, I/O ports and transfer, including epsilon logic. The developer does not have to learn numerous concepts, their relation and exceptions.

Efficiency

The extensive use of variables, functions, tuples and functionals does not affect efficiency of the generated implementation, i.e. these concepts do not introduce any overhead. An ISA developer may be inclined to avoid variables, functions and functionals, as in classical programming languages, their use may have a negative effect on efficiency. However, this is not true for ViDL. The generator statically eliminates all applications of functions and uses of variables. Developers are therefore highly encouraged to use ViDL's functional concepts to improve readability and maintainability.

The same holds for tuples. Even a deeply nested tuple does not cause any overhead in the processor or simulator. Furthermore, if a component of a tuple is not used in the specification, it is eliminated by the generator. In the same way, unused functions and unused variables are eliminated. An eliminated variable resp. function does not allocate any resources, such as functional units in the generated processor.

Functions do not restrain optimizations of the generators. For the function definition

```
fun nand(a,b) = not(and(a,b))
```

the expression `not(nand(a,b)` is transformed into the equivalent expression `and(a,b)`. Optimizations can therefore be regarded as inter-procedural.

Common subexpressions are identified by the generator and respective resources are only instantiated once. This also hold true for subexpressions of different instructions. Hence, the user is not required to factor out common behavior using functions and variables.

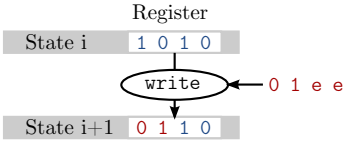


Figure 4.6: Effect of assigning an epsilon expression to a storage.

4.6 Epsilon logic

Instruction sets typically include instructions, which write storages conditionally or partially. For instance, a conditional branch can be understood as a conditional write access to the program counter. Some instruction sets, such as ARM, even define all instruction to be executed conditionally. A store-byte instruction can be regarded as partial write accesses to a 32-bit wide memory. Conditionally executed SIMD instructions may write an arbitrary subset of vector elements of a register. Arithmetic instructions may set an arbitrary set of bits in the status word.

Epsilon logic is a simple and effective method to define such conditional and partial write accesses. It is a unified concept that is powerful enough to define the aforementioned in an intuitive way. The definition is consistent and can easily be changed, compared to the specification of an additional write-enable mask. Basically, epsilon logic is an application of ternary logic, which is a special case of multi-value logic. In ternary logic, a bit may have one of three states. For epsilon logic this is 0, 1 or e, which means “inactive”.

Existing uses of multi valued logic Multi-valued logic (n-ary logic) is a widely known concept. The IEEE standard 1164 [1] for instance defines 9 states for a bit. This logic is also used in VHDL. Bus drivers traditionally use a tri-state logic, where a bit has one of the values 0, 1 or Z (high impedance).

However, VHDL and tri-state logic only focus on buses and do not regard the control of register write enable. Existing ISA specification languages do not seem to utilize a concept like epsilon logic. Instead, separate expressions for the assigned value and for the write enable mask have to be defined in such languages.

Notation Epsilon logic constants can be denoted as literals, such as `0b10ee1`. Besides, the special constant `EPSILON` is defined, which represents an infinite sequence of `e`-bits, similar to the integer `-1`, which represents an infinite sequence of 1-bits.

Assignments Assigning an epsilon logic value to a storage has the effect, that some bits of the storage remain unchanged, as shown in Figure 4.6. An inactive bit (`e`) does not change the state of the corresponding bit in storage. Writing `EPSILON` to a storage does not affect the state at all. To implement this behavior,

the generator decomposes epsilon logic into a value and an enable mask, as described in Section 7.4.2. If an epsilon logic value is written to a port, it is translated into tri-state logic, by mapping the "inactive" state `e` to the "high impedance" state `z`. This enables the specification of typical bus behavior.

4.6.1 Operating on epsilon logic

Epsilon logic values can not only be assigned to storages and ports. They can also be used in expressions. An epsilon logic expression can for example be rotated using the `rotate` primitive. The expression `rotr(0bee11,0b1)` yields the bit-string `rotr(0b1ee1)`. However, some primitives, such as `add`, are not defined on epsilon logic. There is for instance no meaningful interpretation of the expression `add(0b01,0b0e)`.

Basically, all operands that are only subject to bit-permutation may use epsilon logic. This includes the first operand of `cut`, `rotate`, `rev`, `exts` and both operands of `cat`. The alternatives of an `if`-primitive may also use epsilon logic, but the condition must not, as it controls the dataflow. The validity of epsilon logic is checked statically by the generator. If a bit-sensitive operation, such as `add`, is applied on an epsilon logic value, the generator reports an error including its position in the specification.

4.6.2 Review

In the following, the concept of epsilon logic is reviewed with respect to language design criteria (Section 2.3). In particular, its expressiveness is examined.

Expressiveness

Epsilon logic is a simple, yet powerful concept that significantly enlarges the set of specifiable instruction sets. It can be used to specify a wide range of instruction set concepts. This includes conditional execution, bitwise write accesses and narrow store operations (e.g. store-byte).

Conditional SIMD instructions can be defined in a simple way. In the following example `<COND_X>` and `<RES_X>` represent the condition and the written value for the high- and low-part of a 2x16-bit vector.

```
r[d] = cat(
         trunc( if <COND_H> then <RES_H> else EPSILON,16),
         trunc( if <COND_L> then <RES_L> else EPSILON,16)
       )
```

A conditional branch can be considered a special case of conditional execution. The following branch is only triggered, if `cond` holds and the link register `r15` is only written, if the `link` bit is set in addition.

```
pc[0] = if cond then branchTarget else EPSILON
r[15] = if and(cond,link) then add(pc[0],4) else EPSILON
```

So far, a storage has either been written or not. Epsilon logic also allows bit-precise control over write accesses. The following assignment sets only the two most significant bits of a status register.

```
status[0] = 0b10eeeeee
```

ViDL is not limited to the aforementioned special cases. More complex behavior can be specified, by combining conditional execution, partial accesses and operations on epsilon logic. The following conditional "set-bit" instruction sets bit `i` to the value `v`, if `cond` holds.

```
status[0] = if cond then esl(cat(0beeeeeee,v),i) else EPSILON
```

The example also demonstrates the application of an epsilon-shift-left operation (`esl`). The `esl` primitive is a variant of logical-shift-left (`lsl`), which shifts in `e`-bits, instead of zeros. Additional practical examples on epsilon logic can be found in Chapter 6. This section also shows how architectural interfaces and epsilon logic can be combined to define for instance a byte-wise view on word-organized memory.

Reuse

Epsilon logic expressions can be used in combination with ViDL functions without any restriction. The following example shows the definition and application of a function for conditional execution.

```
fun ce(v) = if bit(status[0],7) then v else EPSILON

r[d] = ce(add(r[a],r[b]))
```

The `ce` function encapsulates a specific definition of conditional execution and can be reused among all instructions that are executed conditionally. In the context of a design space exploration, the developer can easily change the condition in `ce`, affecting all instructions consistently. Without epsilon logic, the specification of two separate, but related functions would be necessary.

Portability

The high degree of abstraction also adds to portability. The following ViDL code sets the 4 least significant bits of the 8-bit register `r` to `0101` and leaves the remaining bits unmodified.

```
r = 0beeee0101
```

This behavior is implemented in C and VHDL in completely different ways. In C, it is implemented in a read-modify-write manner, including logical operations for masking.

```
r = (r & 0xf0) | 0x5
```

In contrast, the VHDL implementation defines separate value and mask signals for the register.

```
r(..., value=>"00000101", enable=>"00001111", ...)
```

Thanks to abstraction, both implementations are produced by the generator from the same epsilon logic expression.

Efficiency

Although epsilon logic is a high level concept, it is efficiently broken down to binary logic by the generator. The simulator for instance only evaluates the target of a branch, if the branch is actually taken. This lazy evaluation significantly increases simulation speed (see Section 8.4.5.1).

4.7 Delays

In an instruction set, most instructions have a simple temporal behavior. An instruction immediately reads from registers and writes its result, such that it becomes visible to sequentially executed instructions. However, there are also exceptions to this rule. A load instruction may for instance be delayed. The loaded value does not become visible to the next instruction, but to the next but one. Another example is a branch instruction with delay slots. A delayed branch can be regarded as a delayed write access to the program counter.

As such delays have an effect on program semantics, they must be defined as part of the instruction set. The concept of *delays* in ViDL enables specification of such temporal aspects, while retaining full abstraction from microarchitectural implementation. Using delays, functional and temporal behavior of instructions are decoupled. A delay defines the timing of a read access or a write access. By defining delay-intervals, rather than a specific number of cycles, uncertainty can be modeled. The specification is simple and clear. It abstracts from implementation specific aspects, such as pipeline stages and respective pipeline registers.

In contrast to ViDL, existing specification languages, such as Lisa and nML express temporal behavior indirectly in terms of pipeline registers, bypasses, pipeline stages and their control. Functional and temporal behavior is thereby mixed and can not be regarded separately. Abstraction from microarchitectural aspects is broken, which binds the specification to a specific implementation structure.

Delays in ViDL solve these problems. A delay can be defined for each read and write access. They are also used to precisely define the timing of I/O ports. In the following, the term “vector” is used as an abstraction from storages, ports and architectural interfaces. For each access to a vector, a delay can be defined by two integers $a, b \in \mathbb{N}_0$ with $a \leq b$.

```
r[...]<a,b>
```

(a)

```
instruction A
  ...
  semantics begin
    r[x]<1>   := f(r[x]<0>)
    r[y]<2>   := g(r[y]<0>)
    r[z]<2,3> := h(r[z]<1>)
  end
end
```

(b)

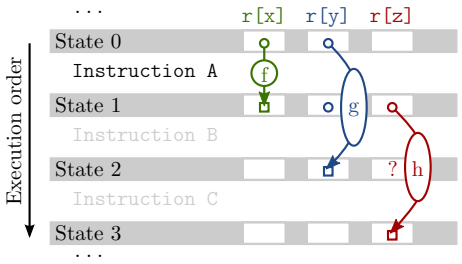


Figure 4.7: Examples of delayed accesses.

The integer a defines the earliest access cycle and the integer b the latest access cycle. If both integers are equal, a single access cycle is defined. This case can be abbreviated as `r[...]<a>`. If instead a is smaller then b, an uncertainty is modeled. In this case, the instruction set does not define exactly, when an access is committed. It only assures, that the access is not committed before cycle a and not after cycle b.

Figure 4.7 shows, the application of delays and their effect on states. The states are enumerated relative to the instruction issue cycle, where state 0 is the state right before instruction issue. The first statement reads from `r[x]` with a delay of 0 and writes with a delay of 1. The instruction therefore maps state 0 to state 1. This is the "normal" behavior of non-delayed instructions. The second statement defines an instruction with one "delay-slot". The result becomes visible in state 2, rather than state 1. A practical example of such an instruction is a branch with one delay slot. The third statement specifies an exceptional timing behavior. A value is captured with one cycle delay from state 1 and becomes visible in state 2 or 3. The interval defines an uncertainty about the cycle in which the result becomes available.

For each vector, a default delay can be defined separately for read and write accesses. If an access does not denote a delay, this default is used. If neither an access delay, nor a vector default delay is defined, an overall delay of 1 is assumed for write accesses and a delay of 0 for read accesses. Hence, "normal" instruction sets without delayed instructions need not define any delays at all. All instructions will map state i to state $i + 1$ according to the overall default delay.

4.7.1 Causality

Roughly speaking, causality means, that an effect is always triggered by a *preceding* event. In the area of digital systems this means, that a state i is only defined by preceding states j for $j < i$. A future state never has an effect on a past state and a state is never defined by itself. Systems in register transfer notation are always causal.

Using ViDL's notation of delays however, non-causal systems can be denoted,

as demonstrated by the following statement

```
r[x]<0> = r[y]<1>
```

Register x of state $s+0$ depends on register y of state $s+1$. In other words, the result is written back, before it can be computed. Such an anomaly is statically detected by the generator and reported to the user. ViDL therefore defines a causality constraint. For each statement, read accesses must precede the write access. For a general statement

```
r[...]<v> = r[...]<u1> ... r[...]<u2> ... r[...]<un>
```

with a write access in cycle v and n read accesses in cycles u_i, the constraint

$$\forall 1 \leq i \leq n : u_i < v$$

must hold. Using this constraint, causality can easily be checked by the generator as part of static semantic analysis.

4.7.2 Review

In the following, the concept of delays is reviewed with respect to language design criteria (Section 2.3). In particular, the simplicity of specifications is regarded.

Simplicity

Delays provide a formalism to directly denoted, the temporal behavior of delayed instructions. Additional variables, FIFOs or registers are avoided, which simplifies notation and increases readability. The following example illustrates, the definition of a delayed branch width two delay slots in an imperative language, in an HDL and in ViDL.

```
Imperative Programming Language:
  simulation_loop
    ...
    pc:=next2
    next2:=next1
    next1:=pc+offs
    ...
  end

Hardware Description Language:
  register reg1(dataIn=>pc+offs, dataOut=>pc1)
  register reg2(dataIn=>pc1, dataOut=>pc)

ViDL
  pc[0]<2>=add(pc,offs)
```

Thanks to the clear notation in ViDL, delayed instructions are directly recognized as such. There is no need to manually analyze instruction behavior and pipeline control to understand the ISA timing. Temporal behavior of existing instruction sets can directly be expressed. In the process of a design space exploration, timing can be adapted in a matter of seconds. For instance, delay slots of branches may be added or removed.

Portability

Delays define the timing behavior in an abstract way. They are not specific to one target domain, such as pipeline registers are specific for a pipelined microarchitecture. Therefore, they can efficiently be mapped to the simulator and to processor implementations with different microarchitectures.

Efficiency

In a microarchitectural implementation, delays are put in effect by the pipeline and its control. Basically, delays lead to omitted forwarding and interlocking. The simulator implements delays in a different way. Only the pipelining effects are simulated, instead of a complete pipeline. The sequential simulation of an inherently parallel pipeline would be extremely slow. Thanks to the abstract specification of temporal behavior, an efficient simulator can be generated, as well as an efficient processor.

The generator may exploit the uncertainty of an access interval to optimize the implementation. Any implementation that commits an access in the defined interval conforms to the ISA. Among all these implementations, the generator can select the simplest one. A specific commit cycle would impose a tighter constraint on the solution.

4.8 Architectural interfaces

Instruction sets typically use structures like architectural register files, virtual address spaces and processor-mode dependent registers. Such structures are basically views on physical storages and I/O ports. It is desirable to specify and use such views directly in an instruction set specification language. For this purpose, ViDL features so called *architectural interfaces*. Architectural interfaces unify the specification of very different instruction set concepts and views. For instance, architectural register files, virtual address spaces and processor-mode dependent registers can be specified.

Figure 4.8 shows an example of an architectural register file that combines 32-bit registers to 64-bit registers. The physical 32-bit registers `r0` and `r1` for instance constitute the architectural 64-bit register `s0`. A write access to `s0` will therefore set the content of `r0` and `r1`. The architectural register file `s` does not have any state. It is just a view on the physical register file `r`. It may also be regarded as an alias to `r`. The architectural register file can be used like a

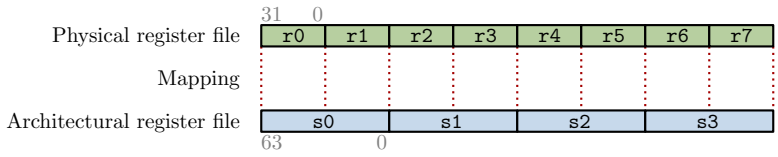


Figure 4.8: Architectural register file.

normal storage. It can be read, written and indexed like a storage of 4 64-bit registers. However, all accesses are delegated to the register file `r`.

In ViDL, the architectural register file is modeled by an architectural interface `s`.

```
storage r
  size  8
  width 32
end

interface s
  size  4
  width 64
  map idx
    s[idx] = cat(r[mul(idx,2)],r[add(mul(idx,2),1)]);
    r[mul(idx,2)]        = cut(s[idx],63,32);
    r[add(mul(idx,2),1)] = cut(s[idx],31, 0);
  end
end
```

The architectural interface is defined by its size, width and mapping.

4.8.1 Mapping

The mapping defines the effect of read accesses and write accesses to the architectural interface. It consists of one read mapping and a set of write mappings. The set contains an arbitrary number of write mappings and may also be empty. The architectural interface `s` defines one read mapping and two write mappings.

Read-mapping The first equation describes the effect of a read access to `s`. It is called the read-mapping in the following. According to the equation, the expression `s[3]` for instance is equal to `cat(r[mul(3,2)],r[add(mul(3,2),1)])`, which is equivalent to the expression `cat(r[6],r[7])`. The value that is read from `s3` is thereby the concatenation of `r6` and `r7`, as shown in Figure 4.8.

Write-mapping The effect of a write access is defined by the remaining equations, which constitute the write-mapping. According to the write-mapping in the example, the assignment `s[3]=0xdeadbeef76543210` corresponds to the assignments

```
r[mul(3,2)]        = cut(0xdeadbeef76543210,63,32)
r[add(mul(3,2),1)] = cut(0xdeadbeef76543210,31, 0)
```

which are equivalent to

```
r[6] = 0xdeadbeef
r[7] = 0x76543210
```

The assignment to `s` is broken down to two assignments. The high part of the assigned constant is assigned to `r6` and the low part to `r7`, as illustrated in Figure 4.8.

Uniform mapping For the given example, the same physical bits are accessed for both, read and write accesses. Technically this means, that the read mapping is the inverse of the write mapping. This can be shown formally, by applying the read and write mappings on the expression `s[x]=s[x]`.

```
  s[x]=s[x]
Apply read-mapping
  s[x] = cat(r[mul(x,2)],r[add(mul(x,2),1)])
Apply write-mapping
  r[mul(x,2)]        = cut(cat(r[mul(x,2)],r[add(mul(x,2),1)]),63,32)
  r[add(mul(x,2),1)] = cut(cat(r[mul(x,2)],r[add(mul(x,2),1)]),31, 0)
Apply algebraic transformations
  r[mul(x,2)]        = r[mul(x,2)]
  r[add(mul(x,2),1)] = r[add(mul(x,2),1)]
```

The last two lines express the identity function, as expected for the combination of a function and its inverse. Read and write accesses to `s[x]` will therefore refer to the same physical bits. In practice, many architectural interfaces have a uniform read and write mapping. This includes partial memory accesses (Section 6.1) and register windowing (Section 6.4). However, there are also meaningful non-uniform mappings, such as the definition of a constant register (Section 6.7).

4.8.2 Review

In the following, the concept of architectural interfaces is reviewed with respect to language design criteria (Section 2.3). Architectural interfaces are very expressive and can be used to define a large set of common storage structures. Specifications of instruction set are simplified and their maintainability is increased.

Expressiveness

Architectural interfaces are a versatile and powerful concept in ViDL that can be used to define a large class of storage and I/O structures. The set of specifiable structures is significantly enlarged compared to existing approaches. Figure 4.9 demonstrates how architectural interfaces are used to model different

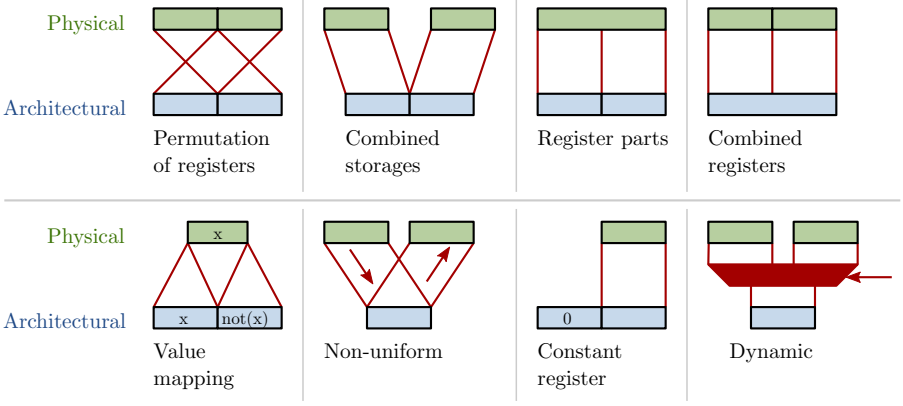


Figure 4.9: Different kinds of mappings between physical and architectural registers.

kinds views on vectors. Views range from static permutation of vector elements to dynamically reconfigurable mappings.

The combination of vector elements shows that an architectural interface is not necessarily mapped to exactly one vector. In general, one architectural interface can be mapped to n vectors. A practical example is the embedding of the program counter into the set of general purpose registers, as for ARM (Section 2.1.1). The extraction of bit-strings and the combination of elements shows that architectural interfaces are not limited to mappings between elements. In general, arbitrary mappings between architectural and vector values can be defined. This enables for instance a normal and a negated view on a value x of a status register. The mapping between architectural interfaces and vectors does not have to be uniform. For instance, two unidirectional physical I/O ports may be combined into one architectural bidirectional I/O port. Another example is the definition of a constant register, which is read as zero, but can not be written, as defined by MIPS (Section 2.1.2). Finally, the mapping between an architectural interface and vectors need not be static. An architectural status register may for instance be connected to different physical storages, depending on the processor mode. A more complex example for a dynamic mapping is register windowing (Section 6.4), which is utilized by SPARC.

Simplicity

Using architectural interfaces, alternative views with meaningful names can be defined, which are then used by instructions. For instance, pairs of 32-bit registers can be combined to architectural 64-bit registers, which are then used by wide instructions. This *simplifies* the specification of instructions and increases readability. The developer can specify wide instructions in terms of 64-bit registers. Wide instructions are immediately identified by the appearance of architectural 64-bit registers. Their semantics are clear, as the developer only needs

to consider architectural 64-bit registers.

Reuse

The abstraction of architectural interfaces also provides a way of *reuse*. Common views are factored out of instructions and encapsulated in an architectural interface. The interface is then reused among all instructions. In the above example, all wide instructions reuse the definition of register pairs. This guarantees a consistent view and thereby also increases *reliability*.

Maintainability

The pairing of registers can easily be modified, without changing touching any instruction. This supports in particular the process of a design space exploration, where rapid and consistent modifications are essential. Another nice example is that of partial memory accesses and their endianness. Byte and halfword views on the memory accesses can be defined by architectural interfaces. These interfaces are then used by the respective load and store instructions. Consistently changing the endianness of an instruction set is thereby a matter of seconds.

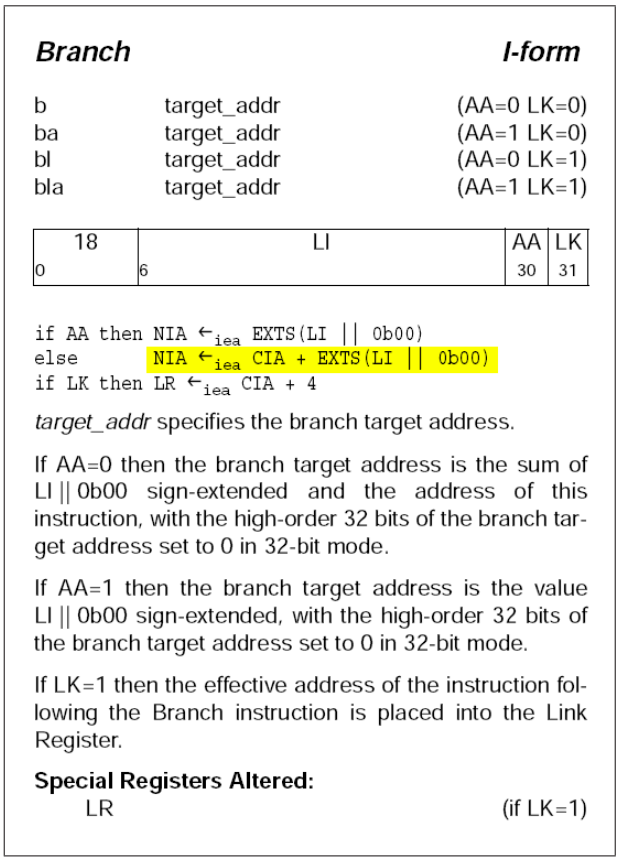
4.9 Type system

Primitives in ViDL operate on bit-strings and only on bit-strings. A bit-string is a sequence of bits with a specific length, such as `010010`. For instance, the expression `cat(10,010)` describes the concatenation of the bit-strings `10` and `010`, yielding the bit-string `10010`. To determine the widths of bit-strings and the width of operations, this section introduces a type-system and a respective type-inference algorithm.

Bit-strings are the only kind of values in ViDL. There are no integers, floating point or string values. This simplifies the language, as the developer has to regard just one kind of value. There is no necessity to define extensive coercion rules between integers, floating point numbers and bit-strings. Although ViDL operates on bit-strings, integer literals may be used in ViDL. These literals actually represent bit-strings of infinite width, as described later.

To implement bit-string expressions in hardware, the width of primitives and intermediate results must be known statically. As this information is not specified explicitly in ViDL, it must be derived from the application context of primitives. For this purpose, the width of bit-strings is modeled using types. Two parameterized types are introduced for exact and minimal widths. The parameter of these types is an integer, which represents the width. Another parameterized singleton type is defined for integer literals. Bit-width rules of primitives are expressed by signatures, which are typically polymorphic. Primitives can therefore be applied to arbitrary bit-widths.

The type of an expression is derived by type inference. Basically, a system of equations and inequalities on parameterized types is solved. The solution of



Branch **I-form**

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)

18	LI	AA	LK
0	6	30	31

```
if AA then NIA ←iea EXTS(LI || 0b00)
else       NIA ←iea CIA + EXTS(LI || 0b00)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Figure 4.10: Definition of branch instructions in the Power ISA manual [26].

the system of equations is a valid assignment of bit-widths to primitives. The type inference algorithm is implemented in the generator and calculates types of a representative instruction set in less than a second.

Running example As a running example, the branch instruction of the Power ISA [26] is used in the following. Figure 4.10 shows the respective excerpt from the ISA manual.

The highlighted expression describes the computation of the branch target address for relative branches. In this expression, the bit-strings `LI` and `0b00` are concatenated (`||`) first. The result is then sign extended (`EXTS`) and finally added (`+`) to the contents of the register `CIA`, which contains the "Current Instruction Address". This sum is then written to register `NIA`, which represents the "Next Instruction Address". The domain of this specification is that of bit-strings, i.e. all operations operate on bit-strings only, not on integers. In the

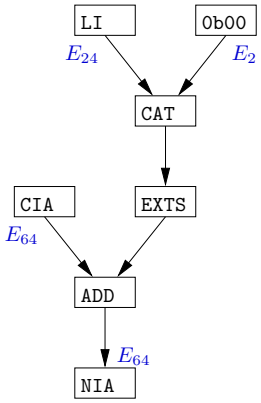


Figure 4.11: Kantorovich tree of the target address computation.

following, the functional notation

$$\texttt{NIA} = \texttt{add}(\texttt{CIA}, \texttt{exts}(\texttt{cat}(\texttt{LI}, 0b00)))$$

is used, to denote the computation of the bit-string expression. The expression is visualized by a Kantorovich tree, as shown in Figure 4.11.

Hardware implementation The Kantorovich tree of the bit-string expression can be used as the dataflow graph (DFG) of a hardware implementation. To implement the DFG in Figure 4.11 in hardware, the width of functions and intermediate results must be known. As hardware is inherently static, these widths must be determined statically. For the `EXTS` function in Figure 4.11, for example, the width of the argument and of the result must be determined. The width of the argument is 26 bits, which is the sum of the widths of the `LI` field (24 bits) and the literal `0b00` (2 bits). As the registers `NIA` and `CIA` are 64 bits wide, the `add` function must operate on 64 bits and hence, the result of `EXTS` must be 64 bits wide. To derive the widths of functions and intermediate results systematically, we use type inference.

4.9.1 Types

The type system that is presented in this section is the result of an extensive exploration on how instructions can be specified clearly, efficiently and unambiguously. Besides, the type system should be simple and understandable on the one hand and powerful enough to specify real-world ISAs on the other hand.

In the following, a set of parameterized types (4.9.1.1), which model bit-width and are in subtype relation (4.9.1.2), is defined first. Using these types, bit-width constraints of functions can be formulated by polymorphic signatures (4.9.1.3). This includes functions that implicitly truncate their actual arguments (4.9.1.4) to support resource sharing. Using a parameterized type for integer

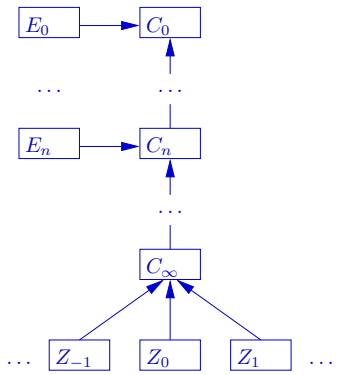


Figure 4.12: Type hierarchy.

literals, even signatures can be expressed that effectively bind arguments to type variables (4.9.1.5). To demonstrate the power of the type system, some representative and non-trivial signatures are finally discussed (4.9.1.6).

4.9.1.1 Bit-string types

The width of a bit-string is represented by a type. Figure 4.12 shows the hierarchy of types used. The types E_x, C_x and Z_x are parameterized types. For E_x and C_x, the type variable $x \in \mathbb{N}_0$ is a non-negative integer. For Z_x, the type variable $x \in \mathbb{Z}$ is an integer.

E_x type: The parameterized type E_x represents the set of bit-strings that are exactly x bits wide.

$$E_x = \{0,1\}^x$$

The set of values of E_2 is for example $E_2 = \{00, 01, 10, 11\}$. Two types E_x, E_y, $x \neq y$ are disjoint and thereby not in subtype relation. The type E_0 contains only the bit-string of length 0, which is the empty word ϵ. As the type contains only a single element, a value of that type carries no information.

C_∞ type: The type C_∞ represents the set of bit-strings with an infinite width. In mathematics, this set is known as Cantor set (denoted $2^{\mathbb{N}}$ or 2^ω) which is homeomorphic to the set of functions $\mathbb{N}_0 \mapsto \{0,1\}$. In the following, an infinite bit-string of C_∞ is represented by such a function, which maps each bit position to a bit value.

C_x type: The parameterized type C_x represents the set of bit-strings that are *at least* x bits wide. The set contains bit-string with a finite width $i \geq x$ and bit-strings of infinite width. The C type is defined using the E type and the

C_∞ type.

$$C_x = \bigcup_{i \in \mathbb{N}_0, i \geq x} E_i \cup C_\infty$$

The set of C_2 contains for example the bit-strings $\{00, 01, 10, 11, 000, 001, \ldots\}$. Note, that the set is infinite.

Z_x **type:** The third parameterized type Z_x is the type of the integer literal $x \in \mathbb{Z}$ and only of this integer literal. Z_x is a singleton set containing the bit-string of the integer x, according to its two's complement representation.

$$Z_x = \{\text{twosComp}_x : \mathbb{N}_0 \mapsto \{0, 1\}\}$$

The bit-string of an integer has an infinite width. The bit-string is therefore defined by a function $\text{twosComp}_x(i)$, which yields the i-th bit of the bit-string. The function twosComp_x is defined with respect to the two's complement.

$$\text{twosComp}_x(i) = \begin{cases} \left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 & x \geq 0 \\ 1 - \left\lfloor -\frac{v+1}{2^i} \right\rfloor \bmod 2 & x < 0 \end{cases}$$

In addition to these bit-string types, types for tuples of bit-strings are defined. The introduction of tuple types simplifies the discussion of the type system.

4.9.1.2 Subtyping

The parameterized types E_x, C_x, Z_x and the type C_∞ are in subtype relation. According to the definition of the C type, C_{x+1} and E_x are subtypes of C_x. In other words, the set of bit-strings of C_x is a superset of E_x and C_{x+1}. As the subtype relation is reflexive and transitive, any type E_y resp. C_y is a subtype of C_x, for $x \leq y$. The subtype relation expresses that wherever a bit-string of *at least* x bits is expected, a bit-string of exact or minimal length $y, y \geq x$ may be given. If a bit-string of *exactly* x bits is expected, a bit-string of exactly x bits must be supplied.

As the type C_∞ contains the bit-strings of infinite width, it is a subtype of any $C_x, x \in \mathbb{N}_0$. In other words: wherever a bit-string of minimal width x is expected, a bit-string of infinite width may be given.

The type Z_x is a subtype of C_∞. The type Z_x contains only the infinitely wide bit-string of the integer literal x, which is also a value of C_∞. In the domain of bit-strings this means that wherever a bit-string of minimal width y is expected, an integer literal may be given instead. In addition, the integer literal x may be given, where the type Z_x is expected. This property of the type system is used in Section 4.9.1.5 to couple arguments and type parameters.

The type C_0 is the top element of the type hierarchy. It represents all bit-strings of at least 0 bits, which is the set of all bit-strings $\{0, 1\}^* \cup \{0, 1\}^\infty$. The type hierarchy does not include a bottom element. The set of values of the bottom type would be empty, as two different E_x types are disjoint.

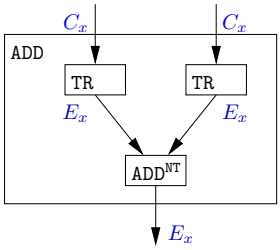


Figure 4.13: Implicit truncation of `add`'s parameters.

4.9.1.3 Signatures

In the previous section, the type hierarchy was introduced, including three parameterized types. These types are now used to define the signature of functions. Most functions are parametrically polymorphic, as they are defined for arbitrary bit-widths. Each function introduces a set of type variables, which are used in parameterized types of the signature. The concatenation function `cat` for instance concatenates a bit-string of exactly x bits and a bit-string of exactly y bits to a bit-string of exactly $x + y$ bits. This is reflected by the signature $\texttt{cat}_{x,y} : E_x \times E_y \mapsto E_{x+y}$, where x and y are type variables of `cat`.

As we use implicit typing, the type variables x and y are inferred from the application context of `cat`. In the example in Figure 4.11, the type of `LI` is known to be E_{24} and the type of `0b00` to be E_2. According to the signature, the result type of `cat` must be E_{26}. This is just a small example to give an idea of the type inference, which is covered in depth in Section 4.9.2.

An important property of implicit typing is that type parameters are inferred and need not be defined by the developer. This is different from explicit typing, where the developer has to specify the values of type variables along with the instantiation. Explicit typing increases the complexity of the specification and reduces the maintainability. This effect is discussed in the evaluation in Section 4.9.3. Implicit typing also allows for optimization of the datapath, but requires an additional analysis phase in the generator.

4.9.1.4 Implicit truncation

To simplify the specification and support the optimization of the hardware implementation, some functions perform an implicit truncation on arguments. A truncation $\texttt{tr}_x(a)$ yields the x least significant bits of the argument a and discards any excessive most significant bits. Figure 4.13 shows an example of implicit truncation. The $\texttt{add}_x$ function truncates the arguments to x bits before the addition is performed. The semantics of the truncating $\texttt{add}_x$ function are defined as

$$\texttt{add}_x(A, B) := \texttt{add}_x^{NT}(\texttt{tr}_x(A), \texttt{tr}_x(B))$$

where $\texttt{tr}_x : C_x \mapsto E_x$ truncates a bit-string of at least x bits to exactly x bits and $\texttt{add}_x^{NT} : E_x \times E_x \mapsto E_x$ represents a non-truncating x bit adder. From

the signature of `tr` and $\texttt{add}^{NT}$, the signature of the truncating addition can be derived as $\texttt{add}_x : C_x \times C_x \mapsto E_x$. To ensure unambiguous semantics of any bit-string expression, the result type is weakened to be C_x, accepting a loss of precision in typing. The rules for unambiguous semantics are discussed in Section 4.9.2.1 in detail. The final signature of `add` is

$$\texttt{add} : C_x \times C_x \mapsto C_x$$

This signature expresses, that the `add` function expects two bit-strings of at least x bits and yields a bit-string of at least x bits.

Implicit truncation is only allowed for functions with certain semantics. The parameter of the `exts` function must for example not be truncated implicitly. This would lead to ambiguous semantics, if the typing is ambiguous. This phenomenon is discussed in detail in Section 4.9.2.1.

4.9.1.5 Coupling of actual parameters and type variables

There are some functions, for which the signature depends on an argument. An example is the $\texttt{ones}(A)$ function, which returns a bit-string of exactly A '1'-bits. The result of $\texttt{ones}(5)$ is for example the bit-string 11111 of type E_5. The type parameter x of the result type E_x is effectively given by the argument A.

As the hardware implementation relies on static typing, the argument of `ones` must be a static value. If the parameter would be dynamic, the result type and thereby the width of the signal would be dynamic.

Exploiting, that a constant argument x has the type Z_x, the signature of `ones` is

$$\texttt{ones} : Z_x \mapsto E_x$$

The type Z_x restricts arguments to integer constants. A dynamic argument will be reported as a type conflict.

Basically, the parameterized type Z_x couples constant arguments with type variables. This way, a value for a type parameter x can be specified explicitly by a (formal) parameter of type Z_x. A nice example to demonstrate the power of this paradigm is the `cut` function. The $\texttt{cut}(V, X, Y)$ function extracts a bit slice from bit X to bit Y from the bit-string V. The application $\texttt{cut}(0b111000, 4, 2)$ yields for example the bit-string 110 of type E_3. The signature of `cut` is defined as

$$\texttt{cut} : C_{X+1} \times Z_X \times Z_Y \mapsto E_{X-Y+1}$$

The type variables X and Y are conceptually defined by the second and third argument in the application of `cut`. These two parameters therefore resemble generic integer parameters. If the type system would not include the parameterized type Z, a concept like generic parameters would be necessary to model the signature of `cut`. However, the combination of parametric polymorphism and the parameterized type Z supersedes the need of generic parameters.

$$
\begin{aligned}
\texttt{add}_x &: C_x \times C_x \mapsto C_x \\
\texttt{cat}_{x,y} &: E_x \times E_y \mapsto E_{x+y} \\
\texttt{extz}_{x,y} &: E_x \mapsto C_y \\
\texttt{exts}_{x,y} &: E_x \mapsto C_y \\
\texttt{rotate}_{x,y} &: E_x \times E_y \mapsto E_x \\
\texttt{cut}_{x,y} &: C_{x+1} \times Z_x \times Z_y \mapsto E_{x-y+1} \\
\texttt{tr}_{x,y} &: C_x \times Z_x \mapsto E_x \\
0ba_{x-1} \ldots a_0 &: \mapsto E_x \\
\texttt{Reg} &: \mapsto E_{\text{width}(\texttt{Reg})} \\
\texttt{RegAssign} &: C_{\text{width}(\texttt{Reg})} \mapsto \text{UNIT} \\
\texttt{Imm} &: \mapsto E_{\text{width}(\texttt{Imm})} \\
< \texttt{IntLit} > &: \mapsto Z_{<\texttt{IntLit}>}
\end{aligned}
$$

Figure 4.14: Signatures of polymorphic functions.

4.9.1.6 Signatures of typical functions

In addition to signatures already mentioned, further signatures of representative functions are listed in Figure 4.14 and discussed in the following. The `extz` (zero-extension) and `exts` (sign-extension) functions for instance extend a bit-string of exactly x bits to a bit-string of at least y bits, where y is independent of x. The result can therefore have an arbitrary width. In the DFG in Figure 4.11, the `exts` function extends a bit-string of exactly 26 bits to a bit-string of at least 64 bits.

A bit-string literal is regarded as a constant function, where the result type expresses the width of the literal. The constant function of the literal $0b00$ has for example the signature $0b00 :\mapsto E_2$. The type C_2 (which is a supertype of E_2) would also be a valid result type, but is less precise and can therefore not be used, where the type E_2 is expected.

A "using" occurrence of a single register in a bit-string expression is also modeled by a constant function. The width of the result is given by the width of the register. The function is therefore not polymorphic. For the CIA register, the signature of the function is $\texttt{CIA} :\mapsto E_{64}$.

An assignment to a register is modeled by a unary function which expects a bit-string that has at least the width of the register. Wider bit-strings are implicitly truncated to the width of the register. An assignment has no result, which is expressed by the result type "UNIT = {()}", as known from SML and other functional languages. For the `NIA` register for instance, the assignment function has the signature $\texttt{NIA} : C_{64} \mapsto \text{UNIT}$.

An immediate instruction operand is modeled by a constant function, whose result type corresponds to the width of the immediate. For the `LI` immediate

$T_1 <: T_2$		T_2			
		C_y	E_y	Z_y	C_∞
T_1	C_x	$x \geq y$	False	False	False
	E_x	$x \geq y$	$x = y$	False	False
	Z_x	True	False	$x = y$	True
	C_∞	True	False	False	True

Table 4.1: Equivalent equations for subtype relations.

in the example, the signature is $\texttt{LI} :\mapsto E_{24}$.

An integer literal x is also modeled by a constant function, where the result type Z_x represents the value of the integer. For example, the signature of the integer literal 4 is $4 :\mapsto Z_4$.

4.9.2 Type inference

The type inference algorithm determines feasible values for all type variables. The values are selected such that the bit-string expression is properly typed. That is the case, if the type of each argument is a subtype of the respective parameter type. Hence, the input for the type inference is a set of constraints, where each constraint is a subtype relation between two types.

For the inference, the set of subtype constraints is transformed into a system of equations, which is then solved by the generator. Table 4.1 shows the translations of subtype constraints into equations, as derived from the type hierarchy in 4.12. For example, the constraint $E_x <: C_y$ is transformed into the inequality $x \geq y$. The constraint $Z_x <: C_y$ always holds and is therefore transformed into the statement "True", which means, that the constraint does not contribute an equation. The constraint $C_x <: E_y$ never holds and is therefore transformed into the statement "False", which means, that the type constraints are contradictory. There is no valid typing of the bit-string expression in this case. The specification is not sound and a respective error is reported to the user.

If there is no contradictory subtype constraint, the transformation yields a system of equations, which is equivalent to the set of subtype constraints. Further inequalities are added to the system of equations, to ensure that the type parameters of E and C types are non-negative. For instance, for an application of $\texttt{cut} : C_{X+1} \times Z_X \times Z_Y \mapsto E_{X-Y+1}$, the inequalities $X+1 \geq 0$ and $X-Y+1 \geq 0$ are added.

This system of equations is then solved by an equation solver. If it has no solution, the bit-string expression can not be typed properly, which means that the bit-string expression is not sound. If it has exactly one solution, the typing is unique. If the system of equations has multiple solutions, the type inference algorithm selects one solution. The proposed theory ensures that the bit-string expressions of all solutions are semantically equivalent (Section 4.9.2.1). The inference exploits this degree of freedom, to optimize the datapath of the bit-string. In particular, the sharing of resources in the hardware implementation

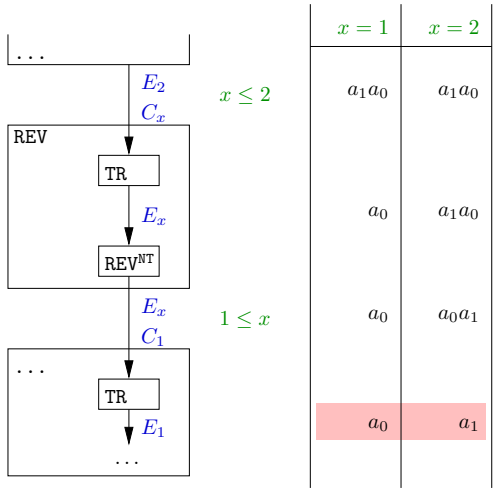


Figure 4.15: An ambiguous typing of the `reverse` function implies ambiguous semantics.

is maximized, by merging common subexpressions in the DFG. An aggressive optimization of the datapath-width may be in conflict with this optimization goal. The next section shows that the semantics of a bit-string expression are equivalent for all solutions.

4.9.2.1 Invariant semantics

The type system may have multiple solutions, in which case the type inference algorithm selects a solution based on an optimization criterion. Which solution is chosen is not defined by ViDL and is therefore undefined from the developer's perspective. The semantics of a bit-string must therefore be the same for all solutions.

The example in Figure 4.15 demonstrates ambiguous semantics for an application of the `reverse` function. The `reverse` : $C_x \mapsto E_x$ function reverses the bits of a bit-string. In the example, an argument of type E_2 is given and a result of type C_1 is expected. To be properly typed, the inequality $1 \leq x \leq 2$ must hold. For the solution $x = 1$, the least significant bit (LSB) of the result is the LSB a_0 of the argument, but for $x = 2$ it is bit a_1 of the argument. As the selection of the solution for x is undefined, the result of the expression is undefined. The origin of this ambiguity lies in the semantics of `reverse` and the implicit truncation of its argument. If the signature is chosen to be `reverse` : $E_x \mapsto E_x$, there is only one solution for x ($1 \leq x = 2$) and the semantics are thereby unique.

In contrast to `reverse`, implicit truncation of the `not` : $C_x \mapsto C_x$ function does not lead to ambiguous semantics because `not` is a bitwise function. In general, solution invariant semantics are guaranteed by two conditions.

- A specific dependence between the type variables of parameter types and the type variables of result types (*E type derivation*).
- A specific relation between the signature and the semantics of a function (*invariant function semantics*).

The resulting invariance of semantics is proven later in this paper.

E type derivation This condition demands that the type parameter of an E_x result is uniquely defined by the E types of the parameters. This informal description of the condition can formally be expressed as follows: For the function f

$$\begin{aligned} f_{x_1,\ldots,x_k} :& E_{g_1(x_1,\ldots,x_k)} \times \ldots \times E_{g_n(x_1,\ldots,x_k)} \\ & \times C_{\ldots} \times \ldots \times C_{\ldots} \mapsto E_{h(x_1,\ldots,x_k)} \end{aligned}$$

and constants a_i, the system of equations

$$\begin{aligned} g_1(x_1,\ldots,x_k) &= a_1 \\ &\vdots \\ g_n(x_1,\ldots,x_k) &= a_n \\ h(x_1,\ldots,x_k) &= b \end{aligned}$$

must have at most one solution for b.

Invariant function semantics The second condition demands that for a given context of a function application, all valid instances of the function must have the same semantics.

$$\begin{gathered} \forall \hat{A}, \hat{B} : \\ \forall \hat{a} \in \hat{A} : \\ \exists \hat{b} \in \hat{B} : \\ \forall f : A \mapsto B <: \hat{A} \mapsto \hat{B} : f(\hat{a}) \simeq_{\hat{B}} \hat{b} \end{gathered}$$

The context of the function is given by the type of the argument $\hat{A}$ and the expected result type $\hat{B}$. For this context, a specific argument $\hat{a}$ must be mapped to the same value $\hat{b}$ by all instances $f : A \mapsto B$, which are subtypes of the expected signature $\hat{A} \mapsto \hat{B}$. The signature is a subtype of the expected signature, if the parameter type is contravariant ($\hat{A} <: A$) and the result type is covariant ($B <: \hat{B}$).

To be precise, the function result and the value $\hat{b}$ need not necessarily be equal. The result and $\hat{b}$ need only be in $\simeq_{\hat{B}}$ relation, which is weaker than the

equal relation.

$$u \simeq_{\hat{B}} v := \begin{cases} \mathtt{tr}_x(u) = \mathtt{tr}_x(v) & \text{for } \hat{B} = C_x \\ u = v & \text{otherwise} \end{cases}$$

If $\hat{B}$ is of type C_x, only the x least significant bits must be equal. That is because the result is truncated by any succeeding function to at most x bits. Any other bits are discarded and need therefore not be equal.

These constraints only regard the signature and semantics of functions. The developer of a primitive must define the signature such, that the constraints hold. The constraints need not be considered by a ViDL user and need not be checked during type inference.

4.9.2.2 Proof of solution invariance

In the following, the proof of solution invariance for all intermediate values is briefly outlined. An intermediate value a_i resp. b_i is solution invariant, if it is a substring of a fixed value a^∞ for all solutions $s \in \mathrm{TSol}$ of the type system.

$$\exists a_i^\infty : \forall s \in \mathrm{TSol} : a_i^s \simeq_{A_i^s} a_i^\infty$$
$$\exists b_i^\infty : \forall s \in \mathrm{TSol} : b_i^s \simeq_{B_i^s} a_i^\infty$$

Solution invariance of E/Z-types In the following, only E types are regarded, but the same applies to Z types. It is first shown that the type parameter of each E-type is solution invariant. This is proven by induction over the topological order of function applications f_i. The induction basis holds, as f_1 is an application of a constant function. The type-parameter x of each E-type result must therefore be constant according to the *E type derivation* condition. For an application f_i, each E parameter is invariant, as the corresponding E result of f_j, $j < i$ is invariant. Since all E parameters are invariant, each E result of f_i is invariant.

Solution invariant subtype T^∞ To show the solution invariance of values, the smallest subtype T^∞ of a given type T is defined first. The type T^∞ is a common subtype of all T^s for $s \in \mathrm{TSol}$, as E/Z-Types are invariant and C^∞ is a subtype of any type C_x.

$$T = T_1 \times \ldots \times T_n$$
$$T^\infty = T_1^\infty \times \ldots \times T_n^\infty$$
$$T_i^\infty = \begin{cases} C^\infty & \text{for } T_i = C_x \\ T_i & \text{otherwise} \end{cases}$$

Solution invariance of intermediate values The solution invariance of the values a_i, b_i is shown by induction over the topological order of f_i. The induction basis holds for a_1, as f_1 is constant and A_1 is UNIT. As a_i is composed of results $b_j, j < i$ which are invariant, a_i is invariant. The next section shows, that b_i is invariant, if a_i is invariant.

Solution invariant result As only a single function application f_i is regarded in the following, the index i of a_i and b_i is omitted. It is proved by contradiction that b is invariant, by assuming that b is not invariant. This means, that b^∞ does not exist. Hence, there must be two solutions $r, s \in \mathrm{TSol}$, such that

$$b^\infty \simeq_{\hat{B}^r} b^r \wedge b^\infty \simeq_{\hat{B}^s} b^s$$

does not hold for any b^∞. Let $\hat{B}$ be the smallest common supertype of B^r and B^s. Then, the values b^r and b^s are not equal with respect to $\hat{B}$.

$$b^r \not\simeq_{\hat{B}} b^s$$

The *Invariant Function Semantics* condition implies, that for a result type $\hat{B}$ all valid function instances f yield an invariant result.

$$\exists \hat{b} \in \hat{B} :$$
$$\forall f : A \mapsto B <: A^\infty \mapsto \hat{B} : f(a^\infty) \simeq_{\hat{B}} \hat{b}$$

As f^r and f^s are functions of type $A^\infty \mapsto \hat{B}$, the results b^r and b^s must be equal with respect to $\hat{B}$

$$\exists \hat{b} \in \hat{B} : f^r(a^\infty) \simeq_{\hat{B}} \hat{b} \wedge f^s(a^\infty) \simeq_{\hat{B}} \hat{b}$$
$$\Rightarrow b^r \simeq_{\hat{B}} b^s$$

This is a contradiction to $b^s \not\simeq_{\hat{B}} b^t$, which was inferred from the assumption that b is not invariant. Therefore b must be invariant. Hence, all values in the DFG are invariant and the semantics of the DFG do therefore not depend on the solution that is selected by the type inference.

4.9.3 Evaluation

In the following, the type system of ViDL is evaluated using the ARM ISA and the Power ISA. First, the branch of the Power ISA is regarded to demonstrate, how to specify non-trivial, real world instructions in ViDL. The specification is simple, compact and intuitive. It is then shown that flaws in the specification such as ambiguous semantics are detected by our type inference algorithm. For instance, a non-obvious ambiguity in the ARM ISA manual has been found. The example of the ARM `add` instruction is used to demonstrate, how implicit truncation increases hardware sharing and reduces the width of the datapath. Finally, the results of the type inference are analyzed, to show that implicit typing and implicit truncation eliminate amounts of type annotations and many applications of explicit truncation.

```
let
  target = if AA then EXTS(CAT(LI,0b00))
                 else ADD(CIA,EXTS(CAT(LI,0b00))),
  NIA    = if M32 then EXTZ(TR(target,32))
                  else target,
  LR     = if LK then ADD(CIA,4) else EPSILON
in
  ...
end
```

Figure 4.16: Specification of semantics of Power ISA branch.

Branch of Power ISA in ViDL Figure 4.16 shows the specification of a real world example in the ISA specification language ViDL. The two bit-string expressions specify the behavior of the Power branch instruction shown in Figure 4.10, including relative and absolute branches in 32 and 64 bit mode, as well as the optional setting of the link register.

Figure 4.17 shows the resulting DFG of the example, annotated with all inferred types. Signatures are selected such, that common subexpressions in the code in Figure 4.16 can be merged in the DFG. The example demonstrates, how easily a complex real-world instruction can be described in ViDL. The complexity of width assignment, implicit truncation and hardware sharing is moved from the user to the type inference of the generator.

Ambiguous semantics in ARM manual Figure 4.18 shows the semantics of the MSR instruction, as specified in the ARM ISA manual. The highlighted expressions describe the rotation of an 8 bit immediate, followed by an extraction of bits 31 to 24. The result of the rotation must therefore at least be 32 bit wide.

Although the semantics of the expression seem to be sound, the application of the rotation is not well defined. The width of the argument suggests an 8 bit wide rotation, whereas the expected result suggests a 32 bit wide rotation. In both cases, either the argument or the result must be extended. Which kind of extension should be applied (sign or zero extension) is not specified. Two aspects of the instruction are therefore undefined: The width of the rotation and the kind of extension.

In ViDL, the combination of both expressions is denoted

$$\texttt{cut}(\texttt{rotate}(\texttt{imm8}, \texttt{cat}(\texttt{imm4}, 0b0)), 31, 24)$$

The ambiguity of this expression is detected by type inference. According to the signatures in Figure 4.14, the result type of `rotate` is E_8, but the expected type for the first parameter of `cut` is C_{32}. As E_8 is not a subtype of C_{32}, a type error is reported. This notifies the designer to correct the expression by inserting a proper sign extension. The following expression precisely defines the

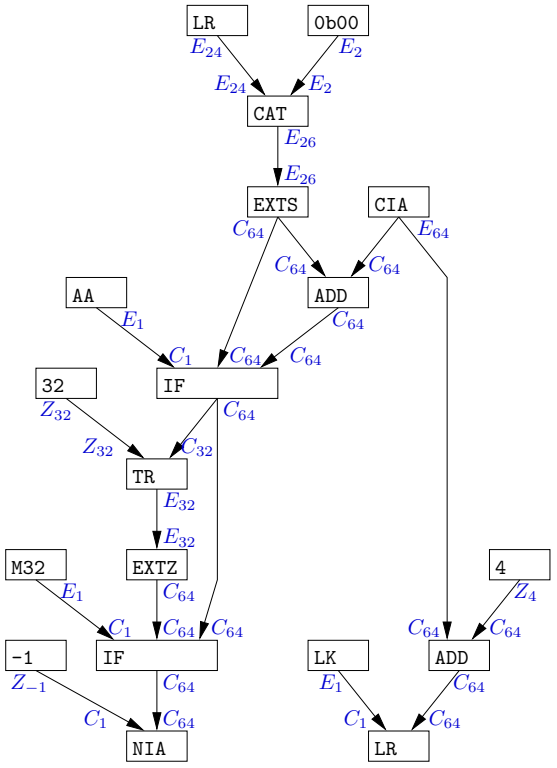


Figure 4.17: Optimized and typed DFG of Power ISA branch.

semantics, as described in an informal text of the ARM manual.

$$\texttt{cut}(\texttt{rotate}(\texttt{tr}(\texttt{extz}(\texttt{imm8}), 32), \texttt{cat}(\texttt{imm4}, 0b0)), 31, 24)$$

Type analysis on the ARM ISA The ViDL specification contains about 40 explicit truncations to uniquely define instruction semantics. The front-end of the generator transforms the ViDL specification into a DFG, on which the following results are based.

The DFG consists of about 1500 function applications, which introduce about 1600 type variables. About 20% of all function applications contribute no type variable, 54% contribute one type variable and 26% contribute two variables. On average, each function application contributes one type variable. The type inference phase of the generator constructs and solves the equation system in less than two seconds.

The 1600 derived type variables define about 4550 types (approx. 3 types per function) in the DFG, of which 60% are C types, 29% are E types and 11% are Z types. The percentage of C types in the DFG is remarkably high, which facilitates the application of implicit truncation in the DFG.

Operation

```
if ConditionPassed(cond) then
    if opcode[25] == 1
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else /* opcode[25] == 0 */
        operand = Rm
    if R == 0 then
        if field_mask[0] == 1 and InAPrivilegedMode() then
            CPSR[7:0] = operand[7:0]
        if field_mask[1] == 1 and InAPrivilegedMode() then
            CPSR[15:8] = operand[15:8]
        if field_mask[2] == 1 and InAPrivilegedMode() then
            CPSR[23:16] = operand[23:16]
        if field_mask[3] == 1 then
            CPSR[31:24] = operand[31:24]
    else /* R == 1 */
```

Figure 4.18: Excerpt from ARM ISA manual [2], page A4-62.

There are 2200 positions in the DFG, where implicit truncation may legally be applied. A real implicit truncation, where the bit-string is actually narrowed, is applied at 140 positions (6.6%) in the DFG. Figure 4.19 shows the number of implicit truncations for the width before and after truncation.

The frequent truncation from infinite bit-strings (INF) to 32 bit is a result of using integers, where a bit-string of at least 32 bits is expected (e.g. `eq(register, 0)`).

The implicit truncations from 33 bits to 32 bits are a result of resource sharing. An `add` instruction for example includes the following two definitions.

```
carry_flag  = bit(add(a,b),32)
gp_register =     add(a,b)
```

Due to resource sharing, only one adder is instantiated, which adds `a`, `b` and yields a 33 bit result. The most significant bit of the result is assigned to the carry flag. For the assignment to the GP register, the 33 bit result is implicitly truncated to 32 bits. Further 33 bit computations that result in implicit truncation include subtraction, left-shift, rotation and the addressing modes of the second operand.

Figure 4.20 shows the distribution of implicit truncations among functions. The 4 most prevalent contributors are the `eq`, `branch`, `cut` and `bit` functions. For the `eq` : $E_x \times C_x \mapsto E_1$ function, the second parameter is implicitly truncated to the width of the first parameter. The `branch` function is a two-way multiplexer, which implicitly truncates parameters of different widths. For the `bit` and `cut` functions, the occurrence of implicit truncation expresses, that the argument is wider than required for the extraction.

Exploration of ISA width To demonstrate, that implicit typing increases the maintainability of the specification, the ARM ISA has been generalized from 32

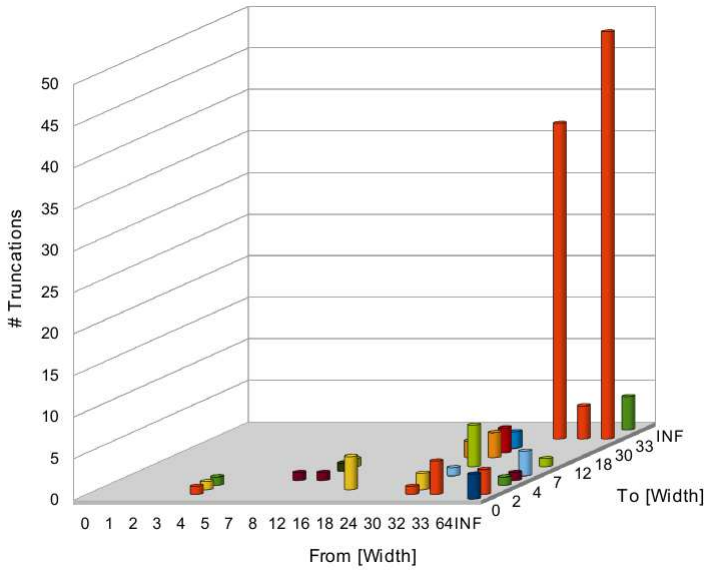


Figure 4.19: Number of implicit truncations from a given bit-width to a given bit-width.

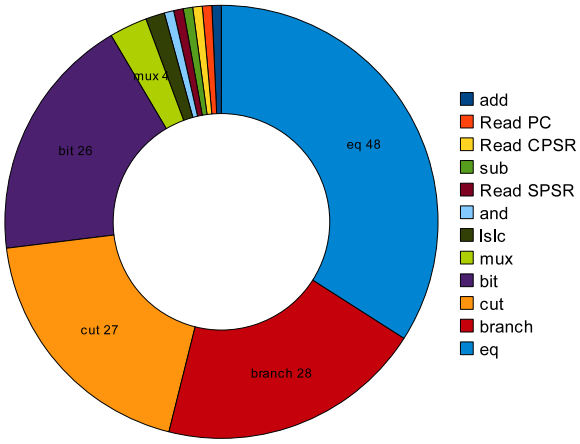


Figure 4.20: Distribution of implicit truncations among functions.

bit to $n \geq 32$ bits. For instance, 64, 65 and 256 bit wide implementations of the ARM processor have been generated. To change the width of the ISA, only the width of the general purpose register file needs to be changed. The width of the datapath (including the shifter-operand) is adjusted automatically, as it is inferred by type inference.

To provoke conflicts, the width of the program counter, status register and memory has not been extended. Therefore, instructions that transfer data between these 32-bit storages and the n-bit general purpose registers have to be considered. Fortunately, these transfers are identified and reported by the type inference as type errors. For the ARM ISA, this is the "branch and link" instruction, the MSR[3] instruction and the load instruction.

To define the most significant bits of the wider general purpose register, a zero-extension is applied. After the insertion of these extensions the width of the ARM ISA can be widened as desired in an instant, by redefining the width of the general purpose registers. For the special case $n = 32$, the extensions have no effect.

Thanks to implicit truncation, transfers from an n-bit general purpose register to a 32 bit register need not be considered. The n-bit value is automatically truncated to 32 bits.

[3]MSR copies the content of the Status registers CPSR and SPSR to a general purpose register

Chapter 5

Transfer primitives

ViDL uses primitives, such as `add` or `exts` to define the semantics of instructions. The primitives itself are defined in a library, using the transfer primitive specification language (TPSL). The definition of a primitive in TPSL includes its signature and its semantics. Semantics are defined separately for each domain (simulator and processor) and each target language, such as C or VHDL. A primitive that is specified in TPSL automatically appears as a polymorphic function in ViDL. From the user's point of view, there is no difference between ViDL functions and TPSL primitives.

The purpose of TPSL is twofold. First, primitives in ViDL are decoupled from target languages. Second, the set of primitives can easily be extended. The generator need not be modified, as the library of primitives is loaded dynamically. The library can be extended in two dimensions, as shown in Figure 5.1. Further primitives can be defined and all primitives can be extended by another target language. This is a major difference to related approaches, which define primitives as part of the specification language. The set of primitives is thereby fixed and typically hard-coded in the generators. The library can be regarded as a planned extension point for ViDL.

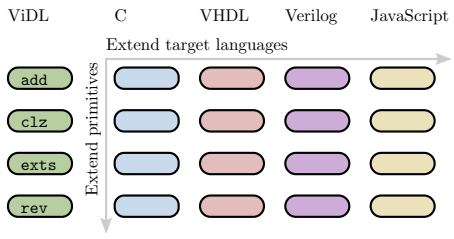


Figure 5.1: Extension of primitive library.

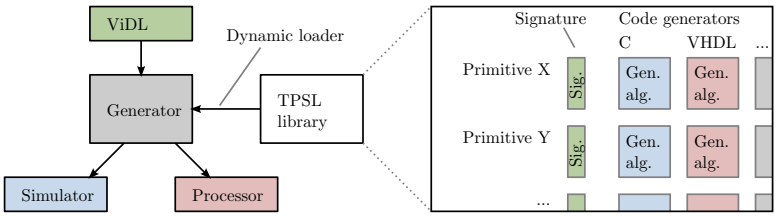


Figure 5.2: Relation between generator and primitive library (TPSL).

5.1 Library

The set of all primitives constitutes the primitive library, as shown in Figure 5.2. The library can be regarded as a layer of abstraction, to separate the specification language from target languages. The example in Figure 5.2 assumes the target languages C and VHDL of the simulator and processor domain.

In contrast to related approaches, primitives are not hardwired in the generator. Instead, the library of primitives is dynamically loaded by the generator on every invocation. Extending the set of primitives does therefore not require recompilation of the generator. This enables a rapid exploration of different primitive implementations, as changes become immediately effective. Different versions of the library may coexist. Each library may use a different implementation scheme for primitives. One library may for instance map primitives to standard VHDL code, whereas another instantiates IP[1] blocks of a certain supplier. The developer can then select one of these libraries at generation time.

The interface between the generator and the library is clear. A developer of primitives need not know about the generator's internals. The capabilities of ViDL can be extended, even if the generator's source code is not published. If primitives would instead be integrated directly in the generator, an extension would typically require access to the generator's source code and recompilation.

An important feature of the primitive library is abstraction from instruction sets. The same library can be reused among all ISA specifications. The expert knowledge that is encapsulated in the definition of primitives is heavily reused. An extension or enhancement of the library directly affects all legacy and future ISA specifications. Currently, the library contains approximately 50 primitives for the target domains C and VHDL. The library is quite comprehensive and extension should be exceptional.

5.2 Primitive

Each primitive in the TPSL library is defined by a signature and a set of code generators. For each target language, one code generator is defined. The example in Figure 5.3 defines an `and` primitive, which conjugates two bit strings. The signature expresses, that "`and`" is a polymorphic function, which operates

[1] Intellectual Property

```
primitive and
  signature C(x) * C(x) -> C(x)
  generator c
    for s in range(slots):
      res[s] = "(%s & %s)" % (param0[s],param1[s])
  endgen
  generator vhdl
    code = "%s <= %s and %s " % (res,param0,param1)
  endgen
end
```

Figure 5.3: Definition of a primitive in TPSL.

on bit-strings of arbitrary width `x`. The primitive defines a code generator for the target language C and another for VHDL. The code generation algorithms are defined in the embedded language Python. Python is a general purpose scripting language, which provides convenient control structures and powerful string operations. It is therefore well suited to define code generation algorithms. The ability to specify generation algorithms is necessary, as the target code typically depends on the actual width of a primitive. A pattern based approach, is therefore not applicable. In Figure 5.3, the `slots` variable denotes the number of integer values used to represent the `x`-bit wide bit-string. This value depends on the primitive's width and the width of integers (e.g. 32 or 64 bit). The latter can be selected by the user at generation time.

$$\texttt{slots} = \left\lceil \frac{\texttt{x}}{\text{intWidth}} \right\rceil$$

The generator then produces a respective sequence of C-statements to implement the `add` primitive.

The C and VHDL generators use different interfaces, as the target domains are quite different. The simulator represents a bit-string as a sequence of integers, which are called "slots". The C-generator therefore has to produce code for each slot of the result. In the example, the result slot `res[s]` is computed by applying C's bitwise `&`-operation on the respective parameter slots `param0[s]` and `param1[s]`. The VHDL-generator in contrast yields only one signal assignment, which uses the `and` operation of VHDL to conjugate the parameter signals.

VHDL-Signals and C-Variables for the result are automatically created by the generator as needed. The same holds for masking of C-expressions. The generator automatically applies masking on excessive bits, which are undefined by convention.

State of primitives

Transfer primitives are required to be stateless. ViDL strictly divides state and transfer of an instruction set to improve clearness of specifications (Section 4.3).

```
uint32 res = param;
res = ((res&0xaaaaaaaa) <<  1) | ((res&0x55555555) <<  3);
res = ((res&0xcccccccc) >>  2) | ((res&0x33333333) <<  2);
res = ((res&0xf0f0f0f0) >>  4) | ((res&0x0f0f0f0f) <<  4);
res = ((res&0xff00ff00) >>  8) | ((res&0x00ff00ff) <<  8);
res = ((res&0xffff0000) >> 16) | ((res&0x0000ffff) << 16);
```

Figure 5.4: Implementation of bit-reverse in C.

As transfer is composed of primitives, a primitive must not carry any state. This constraint must be obeyed by the code generators. For instance, generated C-code that uses a static integer variable would violate this constraint. Actually, this convention is not a limitation in practice, as transfer primitives are likely stateless by their nature.

5.3 Generic primitives

Primitives are versatile, as their code generators can handle arbitrary bit-widths. An example is the C-code generator of the `and` primitive, which can produce code for any bit-width. The generator breaks a wide `and` operation in ViDL down to a sequence of "`&`" operations in C. The generators for such bitwise primitives are quite simple. However, generation algorithms of other primitives are much more complex. For instance, try to write a program that generates *efficient C-code* that multiplies an n-bit integer width an m-bit integer using k-bit integers, with $n, m, k \in \mathbb{N}$. The primitive library includes such a generator.

Complexity of C-code

The code generators for VHDL are typically simple. In most cases, primitives can directly be mapped to generic entities in VHDL. The C-code generators however are more challenging, as they need to break primitives down to standard C integer arithmetic. An example is ViDL's bit-reverse primitive `rev`, which has no direct counterpart in C. A straight forward implementation may use a loop, to assemble the result bit by bit, using shift and mask operations. This solution is linear in the number of bits and therefore slow. ViDL includes a sophisticated code generator, which computes the result in $O(\log(n))$ steps in a divide-and-conquer manner. The generated code in Figure 5.4 demonstrates, how a 30-bit wide bit string is reversed in 5 steps. The example shows, that an optimized C implementation of a primitive may be quite complex. Code generators in the library encapsulate the expert knowledge on how to generate such code.

Optimization of C-Code

To optimize the generated C-code, the primitive generators exploit knowledge from static ISA analysis. The simulation code of a primitive is thereby tailored to

(a) lsr(x,y)

```
lots=width>>32;
offs=y>>5;
shiftr=y&0x1f;
if (shiftr==0){
    for (i=0; i<slots; ++i){
      if (i+offs<slots){
        res[i]=param[i+offs];
      } else{
        res[i]=0;
      }
    }
} else{
  shiftl=32-shiftr;
  for (i=0; i<slots; ++i){
    if (i+offs+1<slots){
      res[i]=param[i+offs+1]<<shiftl|param[i+offs]>>shiftr;
    } else if (i+offs<slots){
      res[i]=param[i+offs]>>shiftr;
    else{
      res[i]=0;
    }
}
```

(b) lsr(x_{64},y_6)

```
param[2]=0;
param[3]=0;
offs=y>>5;
shiftr=y&0x1f;
if (shiftr==0){
  res[0]=param[offs];
  res[1]=param[offs+1];
}
else{
  shiftl=32-shiftr;
  res[0]=param[offs+1]<<shiftl | param[offs]>>shiftr;
  res[1]=param[offs+1]<<shiftl | param[offs+1]>>shiftr;
}
```

(c) lsr(x_{64},32)

```
res[0]=param[1];
res[1]=0;
```

(d) lsr(x_{64},8)

```
res[0]=param[1]<<24 | param[0]>>8;
res[1]=param[1]>>8;
```

Figure 5.5: Customization of primitive implementation in C.

its application context. Figure 5.5a shows the general purpose implementation of a 64 bit `lsr` primitive using 32 bit slots in C. Unfortunately, the implementation is rather inefficient as it includes much control code (marked red), such as loops and conditions. Without information from static analyzes, this is the fastest implementation of the `lsr` primitive in C.

Fortunately, the code generator can produce a more efficient implementation, by using information from static analysis of bit-widths. The result is shown in 5.5b. As the number of slots is known statically, loops can be unrolled, which eliminates expensive control code. In addition, the `param` array can be zero extended, which eliminates a distinction of cases and thereby control code. The optimized code contains only one remaining case distinction. This distinction may seem to be redundant, but actually it is not. The expression `v<<shiftl` is undefined in C for `shiftl` ≥ 32. On Intel processors, the result for `shiftl` $= 32$ is v and not 0, as one may expect. This shows that implementing a primitive in C has its pitfalls and is a non-trivial task. For ViDL, this task is solved by the generator and the developer need not pay attention to the specifics of C.

The code generator can further optimize the C-code in case that the shift distance is constant. Figure 5.5c shows the result for a static 8-bit shift. If the shift distance is a multiple of the slot-width (Figure 5.5d), the primitive can be implemented using only two assignments. Actually, even the assignments are eliminated by the generator. Instead, the expression `param[1]` and the constant 0 are propagated.

Summing up, the examples demonstrate that code-generators produce highly optimized C-code. The customization exploits information from static ISA analyzes, such as bit-widths and constant-parameters. The resulting code is typically free of control structures. It can therefore efficiently be executed on deeply pipelined host processors.

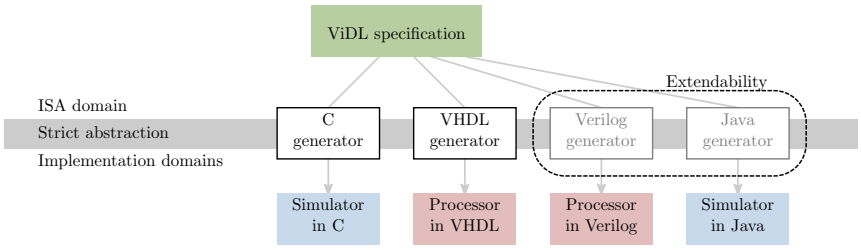


Figure 5.6: Extension by further target domains.

5.4 Review

Reliability is a major concern in processor engineering. Each code-generator defines the mapping of a primitive to one target domain. Of course, the TPSL developer may specify inconsistent generators, such that the resulting implementations behave differently. Therefore, primitives must be tested excessively. Fortunately, primitives are small compared to whole ISAs and do not carry any state, as demanded earlier. Testing can therefore be automated.

Once defined and tested, a primitive can be *reused* in any ISA specification. A primitive is ISA independent and encapsulates expert knowledge on how to implement the primitive in different domains. The examples have shown that an implementation may be rather complex, which confirms the great value of reuse.

The `lsr` example has demonstrated that code generators produce *efficient* implementations. The resulting C-code is typically free of control instructions, which speeds up execution on deeply pipelined architectures. Hence, the ViDL developer does not have to consider special cases and their optimization, but can focus on ISA development. New optimizations can easily be integrated into existing ISA implementations. For instance, an optimization of an existing primitive can be incorporated, by simply regenerating the simulator and the processor. Instruction set specifications need not be touched.

The ViDL generator and the primitives can be *extended* by further target domains, as shown in Figure 5.6. For instance, to add a Verilog target, each primitive is extended by a Verilog generator. After extending the generator as well, each existing ISA specification can be compiled to Verilog code. Again, there is no need to touch any existing ISA specification.

Chapter 6

Design patterns

This section explains how common concepts of instruction sets are elegantly expressed in ViDL. Informal descriptions of instruction sets use various concepts, which can be reduced to a small set of formalisms in ViDL. Typically, multiple orthogonal formalisms of ViDL are combined to resemble a certain concept of an instruction sets. A byte-wise view on a word-wide memory is for instance defined by combining architectural interfaces and epsilon logic. The resulting architectural interface encapsulates the endianness of the instruction set.

In the following, one design pattern is defined for each concept of instruction sets. Some of these design patterns may seem obvious, while others are not suggestive. The patterns have proven to be good practice during specification of ARM, MIPS and Power.

6.1 Partial memory accesses

Many instruction sets define a 32-bit wide data memory and load and store instructions for naturally aligned 32-bit, 16-bit and 8-bit accesses. The latter two obey either a little-endian or big-endian byte-ordering. Such "smaller" accesses can be defined using architectural interfaces and epsilon logic.

In general, an instruction set defines a memory of n words, where each word is m-bit wide. Instructions impose a set of views on the memory, where each

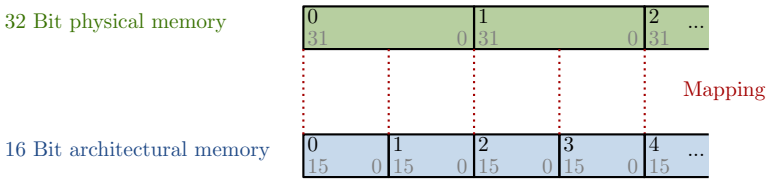


Figure 6.1: A 32 bit wide physical memory and a 16 bit view via an architectural interface.

view divides a word into $2^k, k \in \mathbb{N}_0$ equal sized parts. The view uses either a little-endian or big-endian ordering on the parts.

The following pattern defines a view `<<view>>` on a memory `mem`. The value of `<<e>>` is either `LittleEndian` or `BigEndian`.

```
interface <<view>>
  width <<m / 2**k>>
  size  <<n * 2**k>>
  map index
   mem[omitr(index,<<k>>)] = write<<e>>(view[index],tr(index,<<k>>));
   view[index] = extract<<e>>(mem[omitr(index,<<k>>)],tr(index,<<k>>));
  end
end
```

The functions `writeBigEndian` and `writeBigEndian` use epsilon logic, to write only one part of a memory cell. The address of the memory is the address of the view with the k least significant bits removed. This corresponds to an integer division by 2^k.

The pattern has the following properties: The mapping defined by the architectural interface is uniform. Read and write accesses to the view refer to the same physical bits. The relation between memory bits and view bits can be regarded as a bijective function. This means, that each bit of a view is associated with exactly one bit of the memory. In accordance with bijectivity, the product of `size` and `width` is the same for memory and view.

The following example is an application of the pattern. It defined a 32-bit memory `mem32` of 2^{16} words and defines two views. The first view `mem16` divides the memory into half-words and the second view `mem8` divides the memory into bytes. Both views use a big-endian byte-ordering. A developer may define further views, which use a little-endian byte ordering.

```
storage mem32
   width 32
   size  2**16
end

interface mem16
   width 16
   size  2**17
   map index
     mem32[omitr(index,1)] = writeBigEndian(mem16[index],tr(index,1));
     mem16[index] = extractBigEndian(mem32[omitr(index,1)],tr(index,1));
   end
end

interface mem8
   width 8
   size  2**18
   map index
     mem32[omitr(index,2)] = writeBigEndian(mem8[index],tr(index,2));
     mem8[index] = extractBigEndian(mem32[omitr(index,2)],tr(index,2));
```

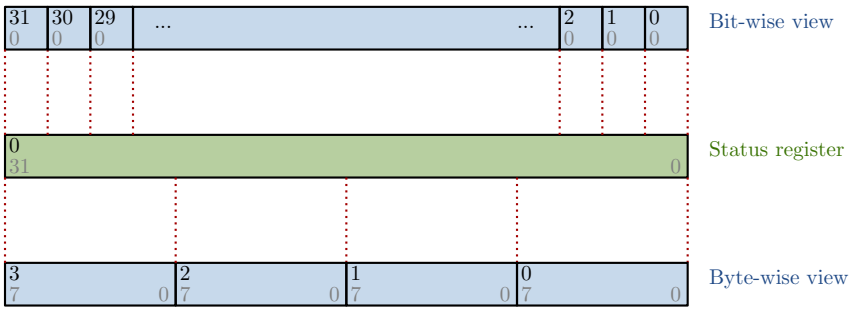


Figure 6.2: Physical status register and additional views via architectural registers.

```
   end
end
```

6.2 Status registers

Most instruction sets define a status register, which contains a set of flags. Instructions typically read and write a subset of these flags. Besides, special *save* and *restore* instructions are typically defined, which transfer the entire status register to a general purpose register and vice versa.

A status register of n flags can be defined in two ways in ViDL. First, it can be defined as a storage of n 1-bit elements and second, it can be defined as a storage of one n-bit element. The first alternative allows random access to the bits of the register. However, this degree of freedom is typically not utilized, but requires a complex implementation. The *save* and *restore* instructions for instance read and write all n flags concurrently, which would results in n read and write ports.

The second alternative of one n-bit element is therefore preferred. It enables reading and writing of the entire status word. Besides, a subset of bits can be written using epsilon logic. To enable simple access to distinct flags or parts of the status word, architectural interfaces are used. Figure 6.2 shows a 32 bit wide status register. The bits of the register can be accessed via the upper architectural register file. For a byte-wise access, a second architectural register file is defined. Actually, the example shows the specification of the ARM status register. The ARM instructions access the status register in a bit-wise, byte-wise and word-wise manner. Thanks to architectural interfaces, these accesses can directly be expressed in ViDL.

The following patterns define a status register of w bits and a view to access v-bit wide parts of the register. The width w is assumed to be a multiple of v.

```
storage status
  width <<w>>
  size  1
```

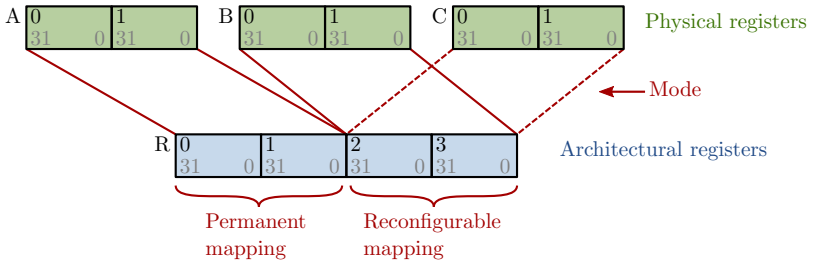


Figure 6.3: Mode dependent mapping between architectural and physical registers.

```
end

interface <<view>>
  width <<v>>
  size  <<w/v>>
  map index
   status[0] = writeLittleEndian(view[index],index);
   view[index] = extractLittleEndian(status[0],index);
  end
end
```

This pattern uses the write functions and extract functions that are also used to define partial memory accesses. The pattern assumes a little-endian ordering, i.e. `view[0]` will refer to the least significant bits of the status register. To define the status register of ARM, the pattern is instantiated for $w = 32$ and $p = 1$ resp. $p = 8$ to define bitwise and byte-wise views.

6.3 Processor-mode sensitive registers

Some instruction sets define dedicated registers for different processor modes. For instance, the register `R14` of ARM represents 6 physical registers, which are associated with 8 processor modes. Depending on the current mode, the respective physical register is accessed.

Such a register structure can be expressed in ViDL using architectural interfaces. Figure 6.3 shows a simple example, which consists of an architectural register file R and three physical register files A, B, C. The architectural registers R0 and R1 are permanently mapped to A1 and A2. In contrast, the architectural registers R2 and R3 are mapped to either B0 and B1 or C0 and C1, depending on the mode.

Figure 6.4 shows the corresponding ViDL implementation. The variable `mode` yields the current processor mode and is typically bound to a specific bit of the status word. The first line of the mapping defines the association for reading accesses. Depending on the mode, either a register of `B` or `C` is mapped

```
storage A, B, C
   width 32
   size  2
end

interface R
   width 32
   size  4
   map index
     R[index] = if mode then C[tr(index,1)] else B[tr(index,1)];
     B[tr(index,1)] = if not(mode) then R[index] else EPSILON;
     C[tr(index,1)] = if mode      then R[index] else EPSILON;
   end
end
```

Figure 6.4: ViDL implementation of mode dependent registers.

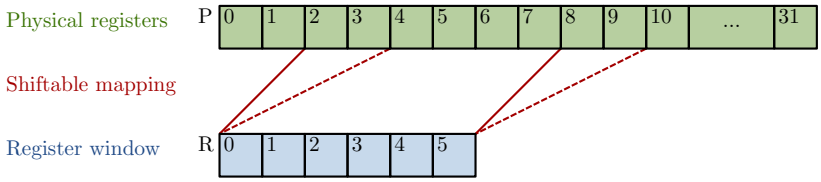


Figure 6.5: Register windowing.

to R. The index is truncated, as the elements 2,3 of R correspond to elements 0,1 of B and C. The other two lines specify the mapping for writing accesses. Depending on the mode, either a register of B or C is written, which is expressed using epsilon logic. For a given mode, the mapping is uniform, i.e. the read and write accesses refer to the same physical register.

6.4 Register windowing

Register windowing is a technique to access a large set of physical registers via a small window of addressable registers. The window can be shifted to change the set of accessible physical registers. Register windowing is for example used by the SPARC architecture [49].

Figure 6.5 shows an example of register windowing, where physical registers can be accessed via a window of 6 registers. The solid red line shows the current mapping and the dashed line the mapping after a right shift by 2 registers. In general, register windowing is defined by the width of registers (r), the size of the window (w), the number of physical registers (2^p) and the distance of shifts (2^d). The following pattern uses these parameters to specify register windowing in ViDL.

```
storage P
```

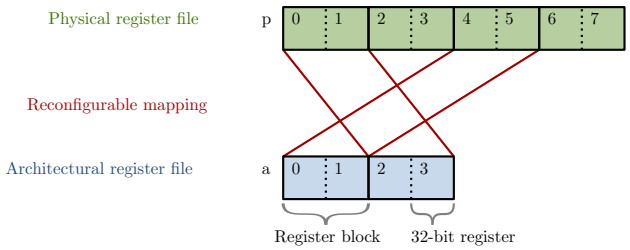


Figure 6.6: Example of a dynamically reconfigurable register file.

```
   width << w >>
   size  << 2**p >>
end

storage pos
   width << p-d >>
   size  1
end

interface R
   width << w >>
   size  << r >>
   map idx
     R[idx] = P[add(mul(pos[0],<< 2**d >>),idx)];
     P[add(mul(pos[0],<< 2**d >>),idx)] = R[idx];
   end
end
```

The mapping of the architectural interface is uniform, i.e. read and write accesses refer to the same physical storage. The register `pos` defines the current position of the register window. The window is shifted by incrementing and decrementing this register by one. In case that the window stretches beyond the end of physical registers, it wraps around to the beginning. The sequence of physical registers can therefore be imagined as a ring. The "shift-window" instruction can also be defined to save and reload content of physical registers, in case of a wrap around.

6.5 Dynamically reconfigurable register files

A dynamically reconfigurable register file [11] consists of a set of physical registers and a smaller set of architectural registers for accessing physical registers. It can be considered a generalization of register windowing. An example of a dynamically reconfigurable register file is shown in Figure 6.6. Architectural registers are mapped block-wise to physical registers. The mapping can be changed dynamically using reconfiguration instructions. The instructions only refer to architectural registers. As their number is smaller than the number of physical

registers, fewer bits are required to encode register operands. Hence, a large set of registers can be utilized, while retaining a tight encoding at the expense of additional reconfiguration instructions.

The following design pattern assumes a dynamically reconfigurable register file of w bit wide registers, 2^b registers per block, 2^p physical blocks and 2^a architectural blocks. This makes 2^{b+p} physical registers and 2^{b+a} architectural registers. The parameters w, b, p and a are natural numbers, i.e. the block size and the number of blocks must be a power of two.

```
storage p
  size  << 2**(b+p)  >>
  width << w >>
end

storage cfg
  size  << 2**a >>
  width << p >>
end

interface a
  size  << 2**(b+a) >>
  width << w >>
  map idx
    a[idx] = p[cat(cfg[omitr(idx,<<b>>)],tr(idx,<<b>>)) ];
    p[cat(cfg[omitr(idx,<<b>>)],tr(idx,<<b>>)) ] = a[idx];
  end
end
```

The storage `cfg` contains the configuration of the actual register file, i.e. the current mapping from architectural to physical blocks. It is indexed with the number of the accessed architectural block and yields the number of the associated physical block. A reconfiguration instruction can change the mapping by assigning the number of a physical block to `cfg`. The mapping of the architectural interface is uniform, i.e. read and write accesses refer to the same physical bits. To implement the example in Figure 6.6, the pattern is instantiated with the parameters $w = 32, b = 1, p = 3$ and $a = 2$.

6.6 Register pairs

Some 32-bit instruction sets define a small number of 64-bit instructions, such as wide multiplications. The 64-bit instructions typically operate on pairs of 32-bit instructions. The specification of these instructions can be simplified, by defining an architectural register file of 64-bit registers, that refer to 32-bit registers. In other words, the i-th 64-bit register is an alias to registers $2i$ and $2i + 1$ of the 32-bit register file. Figure 6.7 shows a physical register file of 8 32-bit registers and an architectural register file of 4 64 bit registers, which are

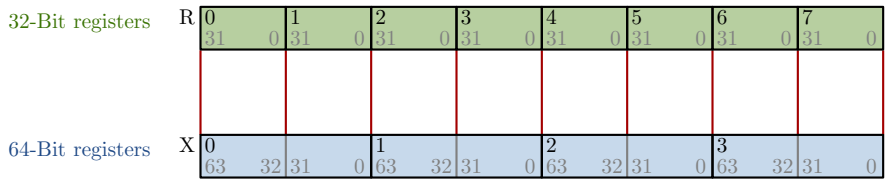


Figure 6.7: Example of a physical register file and a register-pair view.

```
storage R
   width <<w>>
   size  <<s>>
end

interface X
   width <<w*2>>
   size  <<s/2>>
   map index
     X[index] = cat(R[cat(index,0b0)], R[cat(index,0b1)]);
     R[cat(index,0b0)] = cut(X[index],<<w*2-1>>,<<w>>);
     R[cat(index,0b1)] = cut(X[index],<<w-1>>,0);
   end
end
```

Figure 6.8: ViDL implementation of register pairs.

mapped to pairs of the 32-bit registers. Note, that in contrast to the "memory access" pattern and the "status register" pattern, this design pattern combines smaller physical registers to a larger architectural register.

A design pattern for a physical register file `R` and an architectural register file `X` of register pairs is shown in Figure 6.8. The physical register file consists of `s` registers of `w` bits, where `s` is assumed to be even. The mapping between physical and architectural registers is uniform and static. The order of physical registers that constitute an architectural register can be changed, by swapping the constants `0b0` and `0b1`. The register structure in Figure 6.7 is implemented by instantiating the pattern for $w = 8$ and $s = 32$.

6.7 Constant register

Some instruction sets such as SPARC and MIPS define one general purpose register to be always read as 0. The effect of a write access to this register is undefined. As the state of the register is constant, it can be implemented as a constant signal, eliminating the need for a hardware register.

Figure 6.9 shows a register file `R` of 8 architectural registers, where `R0` is always read as zero. The remaining 7 registers of `R` are statically mapped to the physical register file `P`, which consists of 7 registers. Note that the order of

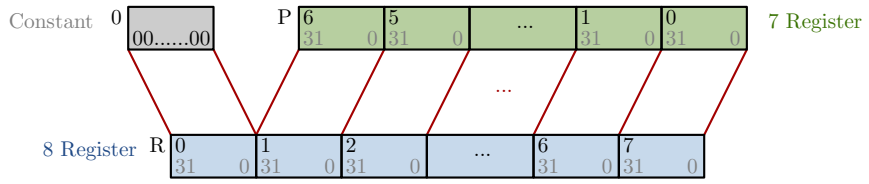


Figure 6.9: Definition of a constant register.

```
storage P
   width <<w>>
   size  <<2**s-1>>
end

interface R
   width <<w>>
   size  <<2**s>>
   map index
     R[index] = if eq(index,0) then 0 else P[not(index)];
     P[index] = R[not(index)];
   end
end
```

Figure 6.10: ViDL implementation of a constant register.

registers in `P` differs from `R`, to allow for a simple mapping. The pattern for a register file `R` with a "constant zero" register `R[0]` is shown in Figure 6.10. It has 2^s registers, where each register is w bit wide. The physical register file `P` contains one register less. The mapping uses a simple bitwise negation, which corresponds to the computation $2^s - \texttt{index}$. As bitwise negation is involutive, it is used for both mappings, physical to architectural and vice versa. Instantiating the pattern with $w = 32$ and $s = 3$ yields the structure in Figure 6.9.

6.8 Embedded program counter

Some ISAs such as the ARM or CoreVA embed the program counter into the general purpose register file. It can be accesses like any other general purpose register, yielding the current instruction pointer (read) or resulting in a branch (write).

In ViDL, the program counter is a dedicated register. It can be embedded into the general purpose registers using an architectural register file, as shown in Figure 6.11. The corresponding ViDL code is shown in Figure 6.12. Note, that a branch is only triggered if register `R7` is written. As the last architectural register is mapped to the program counter, the physical register file need only consists of 7 register.

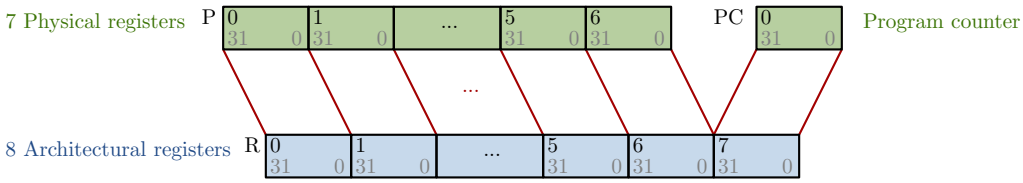


Figure 6.11: Embedding of the program counter in the general purpose register file.

```
storage P
   width 32
   size  7
end

interface R
   width 32
   size  8
   map index
     R[index] = if eq(index,7) then pc[0]    else P[index];
     pc[0]    = if eq(index,7) then R[index] else EPSILON;
     P[0]     = if eq(index,7) then EPSILON  else R[index];
   end
end
```

Figure 6.12: ViDL implementation of PC embedding.

6.9 Branch

In ViDL, the program counter is represented by the dedicated register `PC`. It can be used and behaves like any other register, with two exceptions. First, the register `PC` is automatically incremented by the size of the current instruction, unless it is written by the current instruction. Second, a write access to `PC` sets the address of the next instruction to be executed. Instructions that write `PC` are therefore control flow instructions.

Branch instruction can be classified by different orthogonal aspects. In the following, a small ViDL pattern is given for each aspect. By combining these patterns, a variety of branch instructions can be specified.

Reference of branch target: A branch may be *relative*, *absolute* or *section relative*.

```
/* relative */
pc[0] = add(pc[0],<<offset>>)
/* absolute */
pc[0] = <<target>>
/* section relative */
pc[0] = cat(omitr(pc[0],widthof(<<starget>>)),<<starget>>)
```

The parameter `<<starget>>` specifies the location of the branch within the current section. The size of a section is 2^w, where w is the with of `starget`.

Source of address/offset: A target address (respectively offset) may be given by an *immediate operand* or by a *register operand*.

```
pc[0] = <<immediate>>  /* immediate */
pc[0] = <<register>>   /* register  */
```

The immediate is likely sign- or zero-extended. The pattern assumes an absolute addressing for the sake of simplicity. However, a relative addressing is more likely, especially for an immediate offset.

Condition of branch: A branch may be *conditional* or *unconditional*.

```
/* unconditional */
pc[0] = <<target>>
/* conditional   */
pc[0] = if <<cond>> then <<target>> else EPSILON
```

Note, that a conditional branch is expressed using epsilon-logic.

Additional linking: A branch may store a return address in a *link register*.

```
/* branch */
pc[0] = <<target>>
/* branch and link */
pc[0] = <<target>>
r[15] = add(pc[0],<<instruction size>>)
```

The return address is the address of the instruction that follows after the branch. In case of delay slots, a multiple of the instruction size is added.

Delay slots: A branch may have a specific number of *delay slots*. The number of delay slots basically defines when the branch takes effect. If a branch has n delay slots, the n instruction following the branch are executed, before execution continues at the target address.

```
pc[0]               = <<target>>  /* no delay slot */
pc[0]<  <<d+1>>   > = <<target>>  /* d delay slots */
```

In ViDL, delay slots are defined by a delayed assignment to the program counter. By default, the branch immediately takes effect, i.e. the branch has zero delay slots. Note, that a branch with d delay slots has a write delay of $d + 1$ in ViDL.

6.10 SIMD instructions

A SIMD instruction (Single Instruction Multiple Data) applies the same operation on multiple components of a vector. Vectors are likely stored in general purpose registers. A 32 bit register may for instance hold a vector of 4 8-bit values.

The functional concepts of ViDL allow for a simple specification of SIMD instructions. The following code shows the definition of a two dimensional SIMD addition.

```
let
  (ah,al) = split2(r[a]),
  (bh,bl) = split2(r[b]),
  dh = add(ah,bh),
  dl = add(al,bl)
in
  r[d] = join2(dh,dl)
end
```

The first two lines bind the low and high part of register `r[a]` and `r[b]` to dedicated variables using the `split2` function. The next two lines compute the high and low part of the result, which are then packed using the `join2` function. Using the predefined functional `SIMD2binary`, the specification can further be simplified.

```
r[d] = SIMD2binary(r[a], r[b], add)
```

The `SIMD2binary(x,y,f)` functional interprets the values `x` and `y` as two dimensional vectors. It applies the binary function `f` for each dimension and yields a two dimensional vector. The function argument `f` basically defines the semantics of the processing elements (PEs) of the SIMD instruction. The argument may be a predefined function like `add` or a lambda expression.

Besides `SIMD2binary`, ViDL offers further functionals for unary SIMD functions and functionals for more dimensions. All functionals are polymorphic with respect to the width of the parameters. The `SIMD4unary(x,f)` functional may for instance be applied on a 20-bit wide value, as well as a 64-bit wide value `x`.

6.11 Conditional execution

Some instruction sets like ARM and CoreVA allow instructions to be executed conditionally. Basically, conditional execution enables elimination of control flow instructions by the compiler. For ARM, the execution depends on the value of four status flags (zero, negative, carry, overflow) and a condition-mode. The CoreVA architecture on the other hand defines a register file of condition flags, which control execution. In general, execution of an instruction is enabled by some expression `<<cond>>`. For ARM this is a predicate on the condition flags and for CoreVA it is a condition register.

In ViDL, an instruction is specified to be conditional, by making all assignments conditional. A conditional assignment of a value `<<result>>` to a storage `<<storage>>` is defined using a conditional expression and epsilon logic.

```
<<storage>> = if <<cond>> then <<result>> else EPSILON
```

If the condition holds, the result is assigned to the storage. Otherwise, a sequence of epsilon bits is assigned, which means that the storage remains unchanged. If all assignments are defined this way, the instruction does not have an effect on the processor state.

As conditional execution is typically applied uniformly to all instructions, it is good practice to factor this common aspect out, using a global function. For instance, the following function encapsulates conditional execution as defined by the ARM instruction set.

```
fun ce(cmode,result) =
  let
    cond = switch cmode
             case 0: zFlg;
             case 1: not(zFlg);
             ...
           end
  in
    if cond then result else EPSILON
  end
```

Cases 2 to 14 of the switch statement have been omitted for the sake of simplicity. The parameter `cmode` represents the condition mode, which is encoded in the instruction. Using this function, a conditional addition can be expressed as simple as `r[d]=ce(c,add(a,b))`.

6.12 Complex operand encodings

Some instructions may encode an operand in non-adjacent bits. For instance, the condition mode of CoreVA instructions is encoded in bits 29,28 and 23,20, which results in two fields in ViDL. To refer to the operand as a whole, it is good practice to define a respective variable at the beginning of the semantics section. In general, one variable may be defined for each exceptional operand.

```
semantics
  <<op1>>=<<expr on fields>>,
  <<op2>>=<<expr on fields>>,
  ...
begin
  ...
end
```

The operands `<<op1>>`,`<<op1>>`,... can then be used throughout the specification of instruction semantics. For more complex operands, it is good style to define respective functions to factor out common behavior. The ARM specification for instance uses such a function for mantissa-exponent encoded immediates.

6.13 Addressing modes

In ViDL, addressing modes are expressed as part of instruction semantics. Common addressing modes can be factored out by defining a function, which is then applied in a many instructions. ViDL does not include specialized constructs to specify an addressing mode, which simplifies the language and the implementation of generators.

An *immediate operand* corresponds to a field in the encoding. Its signedness is defined as part of the semantics.

```
r[d] = exts(i) /* signed immediate   */
r[d] = extz(i) /* unsigned immediate */
```

If the immediate is used more than once, it is good style to bind the extended immediate to a variable

```
semantics
  imm=exts(i)
begin
  r[d] = f(..., imm, ..., imm,...)
end
```

Some instructions use a *mantissa-exponent representation* to encode immediates in instructions. For a mantissa m and an exponent e, the represented constant is $m \cdot 2^e$. Such an operand can be defined by `lslc(exts(m),e)`, where m is interpreted as a signed immediate. As an alternative to left-shifting, the mantissa may be rotated (ARM), which slightly extends the set of representable integers.

Some instruction sets define load store instructions with auto-increment functionality. The base register used for addressing is automatically incremented (respectively decremented) by a certain value. The base register may be incremented before (pre-increment) or after (post-increment) computing the memory address. The offset is typically signed and may be encoded in a mantissa-exponent representation. In ViDL, an auto-increment addressing mode can be expressed by an ordinary assignment to the base register. The following example shows the definition of a naturally aligned load-word instruction with pre-increment functionality.

```
semantics
  addr = add(r[a],exts(cat(i,0b00)))
begin
  r[d] = mem[addr]
  r[a] = addr
end
```

Chapter 7

Generators

This chapter discusses the *simulator generator* and the *processor generator*. The simulator generator translates a ViDL specification into an efficient processor simulator, which is implemented in C. The processor generator yields a hardware implementation in terms of VHDL code. Both generators process the very same ViDL specification. The generated products are guaranteed[1] to be semantically equivalent, i.e. there is no need for equivalence testing. The processor generator contributes the entire microarchitecture, including register ports, forwarding and error-prone pipeline control. In the context of a design space exploration, processors with different pipeline structures have been generate and evaluated. As the VHDL Code is entirely generated, it need not be tested. The generated simulator can be used for rapid testing and evaluation of the instruction set specification. Ultra-wide operations in ViDL (e.g. 500 bit multiplication) are automatically broken down to efficient 32/64 bit wide integer arithmetic.

Both generators share the same front-end (Section 7.1) as well as transformation and optimization methods (Section 7.4). The generators only differ in their back-ends, which implement dedicated transformation methods, as described in Section 7.5 and 7.6. Great parts of the implementation are therefore shared by both generators.

7.1 Processing of ViDL

A ViDL specification is first processed by the generator's front-end, which transforms a textual ViDL specification into an intermediate representation. The intermediate representation of instruction semantics is a dataflow graph (Section 7.2). Actually, the generators include two front-ends, one for ViDL and one for TPSL. This section focuses on methods and techniques for the ViDL front-end, which is the considered to be more sophisticated.

The ViDL front-end obeys the classical structure of compilers, as introduced in Section 2.4.1. Great parts of the front-end are generated from specifications

[1] Under the assumption that the generators are correct.

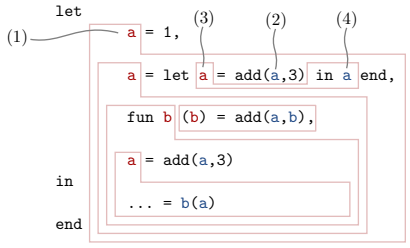


Figure 7.1: Example of scopes in ViDL code.

using the Eli [31] system. One of these specifications is the context free grammar. ViDL has a clear grammar, which consists only of 48 EBNF productions. It belongs to the grammar class *LALR(1)*, which allows for the generation of an efficient bottom-up parser.

Besides lexical and syntactic analysis, the front-end performs name analysis, type analysis and some back-end independent optimizations. The implemented type analysis is discussed along with the type system of ViDL in Section 4.9. Basically, it is used to derive the bit-width of all program objects using type inference techniques. The next sections focus on ViDL's name analysis and some of the front-end's optimizations.

7.1.1 Name analysis

The task of name analysis is to link names to program objects, according to scope rules. ViDL implements basically the scope rules of SML, which are quite intuitive. However, care must be taken to correctly implement name analysis. Scopes may be nested highly recursive, as shown in the artificial example in the Figure 7.1. Although such code is unlikely, name analysis must correctly handle such nesting.

In the example, scopes are indicated by rectangles. A defining occurrence of an identifier is marked red and a using occurrence is marked blue. A definition is valid within the innermost scope, unless it is hidden by another definition in an enclosed scope. For instance, in Figure 7.1 the use (2) of "a" belongs to the definition (1) and the use (4) to definition (3). The definition of the inner scope hides the definition in the outer scope. The example also demonstrates a difference in scope rules between ViDL and SML. In SML, the scope of a function's name includes the body of the function, to enable recursion. As ViDL prohibits recursion, the scope of a function's name does intentionally not include the body.

7.1.2 Optimizations

This section covers some general optimizations, to eliminate for instance redundant definitions. The optimizations are applied by the simulator generator as well as by the processor generator. Examples include the elimination of dead

functions, dead variables, dead parameters and dead tuple components. Note, that additional optimizations (Section 7.4) are performed later on the intermediate representation.

Elimination of dead functions If a function is defined, but not used, it is said to be dead. In such a case, the function can be removed to simplify the generated products. As a result, dead functions do not require any resources. In practice, many functions of libraries are dead. Such functions are automatically identified and removed in an early phase of generation.

Elimination of dead variables If a variable is defined (e.g. using `let`) but not used, it is said to be dead. The expression that is bound to the variable need not be evaluated, as its result is not used. Hardware resources or computation resources for the expression are therefore omitted. Dead variables are automatically identified and eliminated by the front-end. There is no need to manually remove such variables from the specification.

Elimination of dead parameters A dead parameter is a formal parameter of a function that is not used in the function's body. The front-end eliminates such parameters and the respective arguments. Resources for the arguments are thereby omitted.

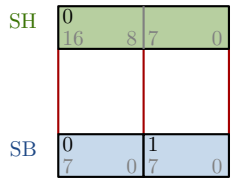
Elimination of dead tuple components If the component of a tuple is not used, it is said to be dead. In this case, the component and its computation can be eliminated. Assume for instance an add carry out function `addco`, which yields the pair of a sum and carry-out. If the carry-out is not used in the application context, the component and its computation are automatically eliminated by the generator.

7.1.3 Translation of architectural interfaces

Architectural interfaces are used in ViDL to abstract from storages and ports. They are intended to improve the quality of an instruction set specification, by increasing the degree of reuse. A developer may use as many architectural interfaces as desired without affecting the complexity of the generated processor or simulator. The implementations are not affected, as the generator translates architectural interfaces into respective dataflow in an early phase of generation. The dataflow is then jointly optimized with the dataflow of instruction semantics. The optimization is typically very effective and eliminates great parts of the dataflow that was introduced by architectural interfaces.

Figure 7.2a shows a simple architectural interface `SB` and its relation to a 16-bit wide status register `SH`. The register can be accessed directly or via the architectural interface `SB`, which consists of the 8-bit wide halves of `SH`. In the adjoining specification of instruction semantics (7.2b), the architectural interface is used in two assignments that set certain bits of the status register. The

(a) Architectural interface



(b) Use in specification

```
...
SB[0] = or(1,SB[0])
SB[1] = or(2,SB[1])
...
```

(c) Translated dataflow

```
...
SH[0] = or(0x0102,SH[0])
...
```

Figure 7.2: Translation of a simple architectural interface.

generator transforms this specification into the representation shown in 7.2c. Both assignments and the `or` primitive are joined to one equivalent assignment to the physical storage `SH`. The transformation considers the specification of the architectural register and applies a variety of algebraic transformations.

The given example is very simple, as it is only intended to demonstrate the principle of architectural interface translation. Actually, the implemented generator translates and optimizes any valid architectural interface. This includes sophisticated applications of architectural interfaces, as presented in Chapter 6.

7.1.4 Analysis of instruction encoding

In ViDL, the specification of each instruction includes the definition of its encoding. The encoding is defined by a *pattern*, which consists of *literal bits* (`0`,`1`), *field* bits (e.g. `d`) and *don't care* bits (`_`), as described in Section 4.4.1. It can directly be copied from instruction set manuals, which use the same notation.

The set of all encoding patterns effectively defines the instruction space. The generator first transforms this set into a decision tree, which is then used to generate the decoder in terms of C- and VHDL-code. The generator constructs the decision tree using the method described by Theiling [51].

The decision tree classifies instruction words, which means that it identifies the instruction of an encoded word. Each node in the decision tree is annotated with a set of *discrimination bits* and a set of *instruction candidates*. Each edge is annotated with a value for the discrimination bits. During decoding, a decision is made at each node, according to the value of the discrimination bits in the instruction word. A leaf node finally represents the decoded instruction. The actual generation of decoder code is described in Sections 7.5.3.

7.2 Intermediate representation

The front-end yields an intermediate representation of the ViDL specification. The intermediate representation describes the structure of storages and the semantics of instructions. The front-end has eliminated redundant parts and substituted architectural interfaces. Besides, ViDL functions have been inlined and variables have been eliminated by copy propagation. As a result, the semantics of all instructions can now be represented by a single dataflow graph (DFG).

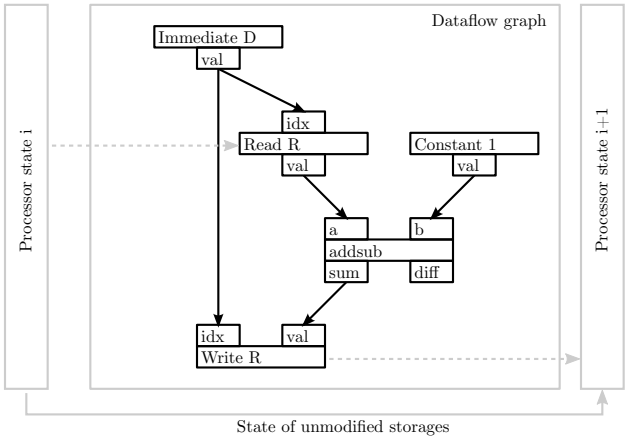


Figure 7.3: Example of a simple dataflow graph (DFG).

A dataflow graph has been chosen, as it is a very basic data structure, which has many advantages in the context of generation. It can easily be analyzed and transformed by the generator. Dedicated algorithms that operate on the DFG are efficient and clear. This reduces the risk of flaws in the generator and thereby improves its reliability. Many transformations can be described concisely as term rewriting rules (Section 7.3). The DFG directly reflects data dependencies between operations. Hence, dead operations are immediately identified and removed. As data dependencies are immanent, the DFG is well suited for code generation. This includes the generation of lazy evaluated C-code (Section 7.5.4).

A DFG consists of operation nodes and their interconnection. The DFG is a directed acyclic graph (DAG), i.e. the dataflow does not include any cycles. Figure 7.3 shows a very simple example of a DFG. It describes the semantics of an "increment" instruction, which operates on a register `r[d]`. In practice, DFGs are much larger. During processing of the ARM instruction set, the number of nodes ranges from 2.000 to 100.000 nodes. An efficient data structure for DFGs is therefore essential.

A DFG can be interpreted as a state transition function, which maps processor state i to state $i+1$. The state is defined as the contents of all registers and all memories of the instruction set (Section 4.2, 4.3). As shown in Figure 7.3, state i is accessed via a read node. The read node yields the contents of a register, i.e. a part of state i. This relation to state i is indicated by a dashed arrow, which does actually not belong to the DFG. The next state $i+1$ is defined by the write node and the current state i. The write node sets the content of a register, i.e. it defines a part of the next state $i+1$. The content of the remaining registers and memories remains unchanged. The respective part of state $i+1$ is therefore defined by state i.

Node types A dataflow graph consists of nodes. A node is either an *internal* node or a *primitive* node. The set of primitive node types is defined by the primitive library (Chapter 5). Besides, 4 different types of internal nodes are defined, namely *Immediate*, *Read*, *Constant* and *Write*. These node types have been identified as the minimal set of internal nodes that are required to define realistic instruction sets. These nodes are called internal, as they require special treatment during generation. The example includes one node of each internal type and a primitive node, namely "addsub".

Ports Each node of the graph has a specific number of so called *in-ports* and *out-ports*. Each in-port is connected to exactly one out-port. Otherwise, the value at that port would not be well defined. An out-port on the other hand may be connected to an arbitrary number of in-ports. The "addsub" node for instance has two in-ports (a and b) and two out-ports (sum and diff). The value of the "sum" out-port is passed to the write node, while the value of the "diff" out-port is unused.

Transfer A node maps values from in-ports to out-ports, where the mapping is specific to the node. The addsub node for instance sets the out-port "sum" to the value $a+b$ and the out-port "diff" to the value $a-b$. It is important to note that nodes in the DFG do not carry any state. As a result, the value at out-ports solely depends on the values at in-ports. This enables numerous optimizations and leads to a clear understanding of processor state. The demand for state-less primitives distinguished ViDL from VHDL, where entities may very well contain registers and therefore carry state.

7.2.1 Instruction DFGs

As mentioned, the dataflow graph combines the semantics of all instructions. The DFG is therefore also called a *global DFG*. The use of a single DFG enables sharing of common subexpressions among different instructions. This significantly reduces the number of nodes and thereby the complexity of the intermediate representation.

However, for some tasks of the generator, the dataflow of single instructions needs to be regarded. In such cases, the generator derives an *instruction DFG* from the global DFG. The instruction DFG is a subgraph of the global DFG and contains all nodes that are relevant for a particular instruction. Basically, those are the instruction's write accesses and their predecessors. For an instruction x, the set of nodes V_x is the minimal fixpoint of

$$V_x = \text{writesOf}(x) \cup \{v \in V | \exists w \in V_x : (v, w) \in E\}$$

The instruction DFG G_x is then the subgraph of G that is induced by V_x. Note, that the sets $V_x, V_y, x \neq y$ may not be disjoint, due to common subexpressions of different instructions.

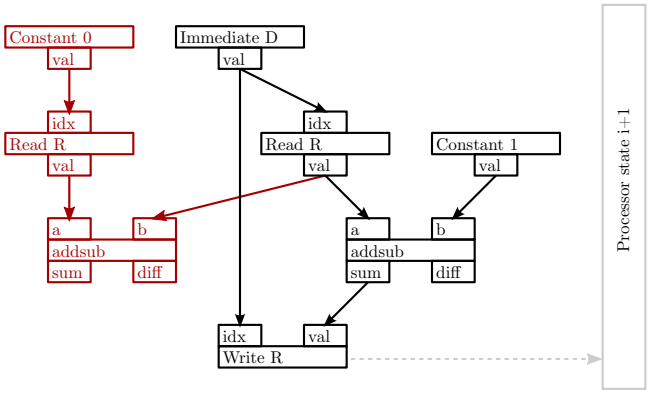


Figure 7.4: Example of a reducible DFG.

7.2.2 DFG simplification

So far, the structure of the DFG and its nodes has been described. This section discusses how the complexity of a DFG can be reduced by removing irrelevant nodes. A node is irrelevant, if there is no path from that node to a write node. In this case, the result of the node does not affect the next processor state and can therefore be removed without affecting the semantics of the DFG. Figure 7.4 shows a simple example, in which the red nodes can be removed. There is no path from these nodes to a write node and therefore, they do not contribute to the next processor state. Irrelevant nodes frequently show up after translations on the dataflow graph. An efficient reduction is therefore essential. The reduction algorithm that is implemented in the generator iteratively removes nodes that are neither write nodes, nor have a successor. Using a queue to keep track of such nodes, the DFG is reduced in $O(|V|)$. The red nodes in the example are removed bottom-up.

7.2.3 Origin information

The ViDL generators annotate each node in the DFG with information about its origin. An "add" node is for instance annotated with the line number of the respective `add(...)` call in the ViDL specification. Origin information basically relates the DFG to the original ViDL specification. This mainly servers two purposes: First, to help the user to *debug* an ISA specification and second, to help the user to *improve* an instruction set.

7.2.3.1 Debugging

To efficiently debug an incorrect instruction set specification, the user relies on meaningful error messages. If the specification includes for instance a type conflict, the ViDL generator emits an error message, which includes the source position of the error. The user can directly fix the specification without tedious

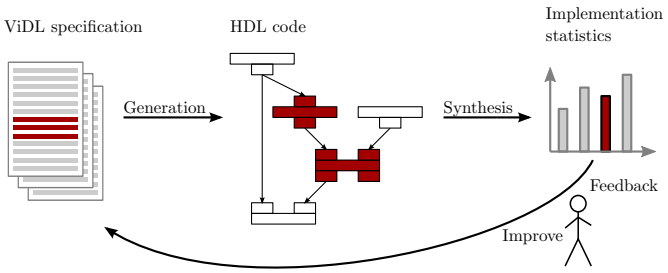


Figure 7.5: Feedback driven optimization of a ViDL specification.

debugging. Some static semantic errors are detected in very late translation phases of the generator. Assume for instance, that a user defines an instruction with dependent load and store accesses to the data memory. Such instructions can not be implemented using a load-store microarchitecture. During synthesis of the microarchitecture, the generator therefore emits an error message, which includes source code references to the read accesses and write accesses.

7.2.3.2 Feedback driven ISA optimization

Assume a scenario where a processor is generated from a ViDL specification and synthesized, as shown in Figure 7.5. The process finally yields a set of implementation characteristics in terms of speed, area and power consumption of the processor. A developer may want to improve these characteristics by changing the instruction set. The critical path of the generated processor may for instance be reduced, by changing the semantics of an addressing mode. To reduce the area or power consumption of a processor, the developer may remove instructions or part of their functionality. To perform such improvements, it is crucial to relate entities of the generated processors to the original ViDL source code. The generator therefore annotates the generated HDL code with source positions of the ViDL specification. A VHDL entity is for instance annotated with an instruction name and the line number of its origin. A developer can then use the annotated feedback from synthesis to optimize the initial instruction set specification.

7.2.3.3 Origin relation

Source positions and nodes in the DFG are actually not associated in a one-to-one manner. For instance, one source position may result in multiple DFG nodes as a result of transformations. On the other hand, one node may be associated with multiple source positions, due to merging of common subexpressions. Entities of the specification that are statically evaluated are not even associated with any node of the DFG. And finally, some nodes that are introduced by the generator are not associated with any source position. The nodes of the forwarding circuit are an example for such nodes.

The association of source positions and DFG nodes is modeled by a binary relation "Origin" on the source positions and the DFG nodes.

$$\text{Origin} \subseteq \text{SrcPos} \times V$$

The relation $Origin(p, v)$ holds if and only if the source position p is associated with the node v. Using the relation, all of the phenomenons mentioned above can be represented.

7.2.3.4 Transformations

During generation, the DFG is massively transformed. Nodes are replaced, merged, deleted and created. During generation of an ARM processor, approximately 100.000 nodes are created and deleted. To maintain origin information, each transformation of the generator needs to updates the Origin relation. For instance, if a node v is replaced by a node w, the relation is updated such that

$$\text{Origin}'(p, w) = \text{Origin}(p, v)$$

holds. If two nodes v, w are merged to a node x, the resulting relation Origin$'$ satisfies

$$\text{Origin}'(p, x) = \text{Origin}(p, v) \vee \text{Origin}(p, w)$$

These simple examples are just meant to give a rough impression. The generator includes transformations which operate on a much larger set of nodes.

All translations in the ViDL generator are implemented to properly maintain origin information. This includes the term rewriting system, which has been extended for this purpose, as described in Section 7.3.3. If only one generation phase would not update the Origin relation, origin information would be lost. Fortunately, the ViDL generator carries origin information through the whole generation process. The implementation of this generator feature has been laborious, but it is crucial for debugging of instruction sets and feedback driven optimizations.

7.3 Term rewriting system

The generator includes a term rewriting system to transform the dataflow graph. It basically implements optimizations and helps to simplify handcrafted transformations (Section 7.3.2). Actually, the rewriting system drives most transformations and optimizations that are explained in Section 7.4.

The term rewriting system has been developed along with the ViDL generator as part of this thesis. It includes 4 important features that are not provided by existing systems. These features are somewhat specific to dataflow graphs and the task of processor generation. In particular, the term rewriting system

- preserves origin information (Section 7.3.3),

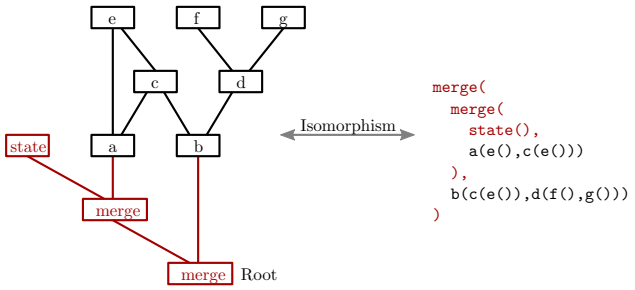


Figure 7.6: Isomorphism between dataflow graphs and terms.

- supports arithmetic and predicates on virtually unlimited integers (Section 7.3.4),
- allows for the definition of labeled sets of transformation rules (Section 7.3.5) and
- includes an extension to define bit-width sensitive rules (Section 7.3.6).

The need for these features and technical aspects have motivated the development of a dedicated term rewriting system. It is seamlessly integrated into the generator, as it operates directly on the dataflow graph. A respective isomorphism between the dataflow graph and terms is discussed in the next section.

7.3.1 Isomorphism

To apply term rewriting on a dataflow graph, the graph has to be modeled as a term. This section describes an isomorphism between dataflow graphs and terms, which is used by the implemented term rewriting system. Each dataflow graph can be represented by a term and vice versa. Basically, a node in the DFG corresponds to an operation and the structure of the DFG matches the structure of the term.

Figure 7.6 shows the correspondence between a dataflow graph and its representation as term on the right. The black subgraph is the actual dataflow graph. The dataflow graph is usually a DAG with multiple root nodes, where each root node is a write accesses (`a` and `b` in the example). These root nodes are combined to a single root node using the red subgraph. As a result, the graph can be represented by a single term. Otherwise, a set of terms would have been needed, one for each write node. There is also a meaningful interpretation of the red subgraph. The `merge(s,w)` operation applies the write access `w` on the state `s` and yields the new state. The constant function `state()` yields the current processor state. The whole term thereby represents the next processor state.

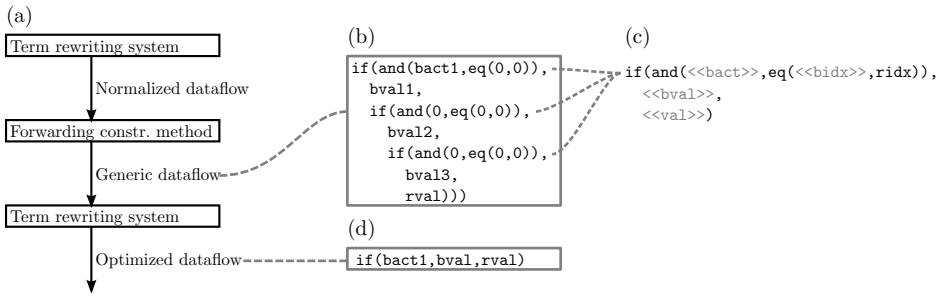


Figure 7.7: Simplification of transformations by term rewriting.

7.3.2 Applications

Term rewriting plays an important role in the generator for two reasons. First, most optimizations are formulated as rewrite rules. For instance, term rewriting is used to evaluate constant expressions (Section 7.4.1).

Second, term rewriting helps to simplify most transformation algorithms, such as the construction of forwarding circuits (Section 7.6.7). Handcrafted transformations benefit in two ways. First, they can assume a normalized dataflow graph, as produced be term rewriting. For instance, all constant terms are represented by their value. Second, handcrafted transformations need not struggle to produce efficient dataflow. Instead, they can produce generic dataflow which typically includes much overhead. Figure 7.7b shows an example of such inefficient generic dataflow, which is produced by the forwarding construction method (Section 7.6.7). The code describes a bypass multiplexer, which is simply constructed by successively instantiating the pattern in Figure 7.7c. For the assumed application of forwarding in the example, the multiplexer's complexity can be reduced significantly. The reduction is performed by term rewriting, which evaluates constant subexpressions. The resulting optimized dataflow is shown in 7.7d.

7.3.3 Origin information

Each node in the dataflow graph is associated with an origin, which is typically a line number of the ViDL specification. During term rewriting, nodes are repeatedly removed and created. In practice, most nodes of the dataflow graph are replaced during rewriting. Applying existing term rewriting systems likely leads to a loss of nearly all origin information.

As mentioned at the beginning of this section, the preservation of origin information has been one of the reasons for the development of a dedicated term rewriting system. The system uses an extended notation for rewriting rules, as shown in Figure 7.8a. The term rewriting rules R1 and R2 are annotated (blue) to transfer origin information from the redex to the contractum. For instance, in the first rule R1, the `not` operation is related to the origins of `sub` and `-1`.

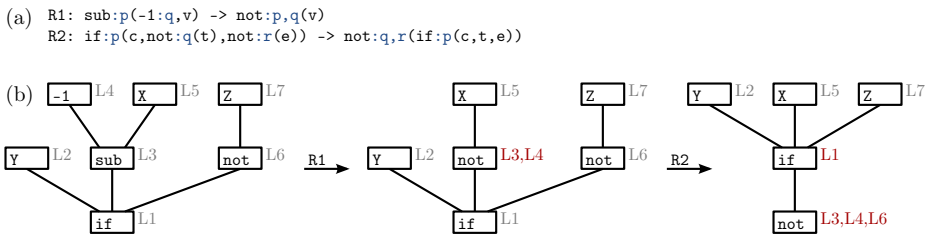


Figure 7.8: Preserving origin information during term rewriting.

The annotation basically expresses the correspondence between operations of the redex and the contractum.

Figure 7.8b shows the application of R1 and R2 on a DFG. Nodes in the DFG are annotated with line numbers `L1` to `L7`. Origin information from the left DFG is entirely preserved. Without the proposed mechanism, the information marked red would have been lost. The origin relation of `if` remained unchanged, while `not` is related to the origins of three nodes that have been eliminated. The example also demonstrates, that after term rewriting, one node may be related to multiple origins. In such a case, multiple ViDL definitions contributed to that node.

The transfer of origin information is optional, i.e. a developer is not forced to annotate a rule. One origin variable can be used for multiple operations in the contractum. In this case, multiple nodes are linked to the same origin.

7.3.4 Integer arithmetic

Many rewrite rules, especially those for static evaluation (Section 7.4.1), need to operate on integers. Assume for instance an application of the `add` primitive on two constant nodes. An evaluation rule for `add` needs to compute the sum of the integers that are associated with the constant nodes.

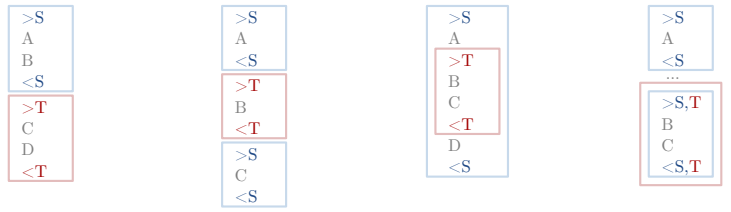
An efficient and unrestricted support for integer computations is therefore crucial. The implemented term rewriting system uses the arbitrary-precision library `gmp` to operate on integers. The precision of these integer operations is virtually unlimited. In particular, integers are not limited by the host width, i.e. 32 or 64 bit. Integer arithmetic is very efficient, since it is directly mapped to `gmp` operations.

The following evaluation rule matches any application of `add` on two constants.

$$\texttt{add}(\text{Const}_a, \text{Const}_b) \rightarrow \text{Const}_{a+b}$$

The integral values of the arguments are bound to a and b. The contractum then constructs a new constant value, which is defined to be the sum of a and b. The sum of a and b is computed using `gmp` functions[2] of virtually unlimited

[2]For the sake of readability, this thesis uses an intuitive mathematical notation instead of GMP function calls



(a) Sets (b) Partial (c) Nesting (d) Set lists

Figure 7.9: Example of rule set definitions.

precision. Note, that the rule only matches static arguments. In contrast, a radix like $\texttt{add}(a, b)$ would match any application of `add`, including those with dynamic arguments, which can not be evaluated statically.

The applicability of some rules does not only depend on the structure of the radix, but also on the value of some integral arguments. For instance, a multiplication can be replaced by a shift, if one factor is constant *and* a power of two. To express such constraints, the term rewriting system features predicates on integer variables that are bound in the radix.

$$\texttt{mult}(x, \text{Const}_a) \quad \boxed{\text{isPow2}(a)} \quad \rightarrow \texttt{cat}(x, \texttt{zeros}(\text{Const}_{\log2(a)}))$$

The rule is only applied, if the radix matches and the predicate holds true. The term $\texttt{mult}(x, 8)$ is for instance translated into $\texttt{cat}(x, \texttt{zeros}(3))$, but $\texttt{mult}(x, 7)$ remains unmodified. The notation of the predicate has been simplified for readability. Actually, it uses `gmp` functions to check the condition.

7.3.5 Rule sets

In some cases, it is desirable to suppress the application of certain term rewriting rules. For instance, the generator includes rules that are beneficial for the generation of processors with long pipelines, but downgrade the quality of processors with short pipelines. The developed term rewriting system therefore includes a mechanism to selectively activate and deactivate certain sets of rules. The mechanism has also been used to evaluate the effects of rewrite rules. Besides, it has proven to be very practical during debugging. A major application of rule sets, namely pipelining of operations, is discussed in Section 7.6.2.

Rule sets are defined along with term rewriting rules, as shown in Figure 7.9. Letters A to D denote rules and S,T represent rule sets. A simple definition of two disjoint sets is shown in (a). The definition of a set may also be split (b) into multiple parts. In the example, S is defined to contain the rules A and C. The definition of sets may be nested (c), to easily define subsets. In the example, S contains rules A to D and T contains rules B and C. Finally, rules can easily be added to multiple sets using the notation in (d). Rule A is contained in set S and rules B,C in set S and T. The latter definition is very practical for the definition of target frequency dependent transformations.

7.3.6 Bit-widths

A surprisingly large number of rewrite rules need to consider bit-widths of subterms. For instance, the bit extraction $\texttt{bit}(\texttt{cat}(x, y), 8)$ can be rewritten into an extraction on x or y, depending on the width of y. If y is 6 bit wide, it can be transformed into $\texttt{bit}(x, 2)$. If y is 16 bit wide, it can be rewritten to $\texttt{bit}(y, 8)$.

To seamlessly integrate bit-widths, the term rewriting system uses an extended notation of rules. In the redex, the width of a term t is bound to a variable a using the notation t/a. The variable a can then be used like a constant that was bound by Const_a. In particular, it can be used in the rule's predicate and contractum in combination with other bit-widths and constants. As a result, the extended term rewriting is very powerful and can express a variety of transformations. For instance, the following rules use the extension to implement the aforementioned optimization

$$\texttt{bit}(\texttt{cat}(x, y/a), \text{Const}_z)\boxed{z < a} \rightarrow \texttt{bit}(y, \text{Const}_a)$$
$$\texttt{bit}(\texttt{cat}(x, y/a), \text{Const}_z)\boxed{z \geq a} \rightarrow \texttt{bit}(x, \text{Const}_{z-a})$$

The width of y is bound to a variable a. The variable is then used in the predicate to decide, which rule to apply. It is also used in the second rule, to compute the bit position within x. As the example demonstrates, the extended notation is intuitive and readable. Currently, the generator includes 260 rewrite rules, among which 38 rules are bit-widths dependent. The notation is therefore crucial to define a clear and simple database of rewrite rules.

7.4 Transformations and optimizations

This section presents transformations on the intermediate representation, i.e. the dataflow graph. They are independent from ViDL as well as from the generated products. Most transformations utilize the term rewriting system. Either a part of a transformation or the entire transformation is implemented using rewrite rules. For instance, *algebraic transformations* and *strength reduction* can entirely be expressed by term rewriting rules.

Another optimization that utilizes term rewriting is *partial evaluation* (Section 7.4.1), which eliminates static parts of the dataflow graph. Besides these optimizations, a so called *epsilon transformation* is presented (Section 7.4.2). Epsilon transformation substitutes epsilon logic by semantically equivalent write-enable expressions. Such expressions are required to implement efficient simulators and processors. The respective code generators therefore expect the intermediate representation to be in this form.

7.4.1 Partial evaluation

A realistic instruction set specification contains many expressions that can partially be evaluated by the generator. These parts need not be evaluated by the simulator and do not require any hardware resources. In practice, there are four

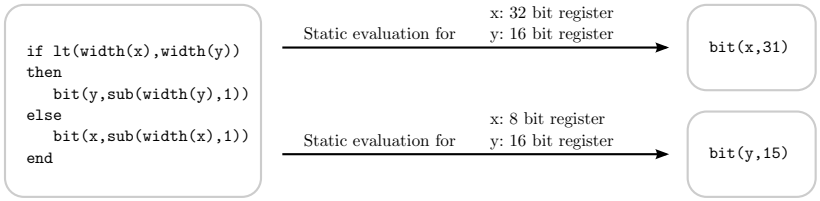


Figure 7.10: Example of static evaluation.

major sources for constant expressions, namely (1) constant fragments of the specification (2) functions that are called with constant arguments, (3) static indexing of architectural interfaces and (4) bit-width dependent expressions.

An example for the latter is shown in Figure 7.10. The example is somewhat artificial, but is well suited for the purpose of demonstration. The expression on the left yields the most significant bit of the values `x` and `y`. For instance, if x is an 8-bit wide register and y is a 16-bit wide register, the expression returns bit 15 of `y`. Since the width of all expressions is determined statically by type inference, term rewriting substitutes the `width` operation by the width of its argument, i.e. 16 resp. 8. In subsequent rewrite steps, `lt`, `if` and `sub` are replaced, which finally yields the expression on the right.

It should be mentioned, that static evaluation operates on integral numbers of virtually unlimited precision. The generator uses an arbitrary precision library. In literature, the term "arbitrary prevision" refers to integer operations that are only limited by the computer's memory. For the generator, this is the range of approximately $-10^{1292913986}$ to $10^{1292913986}$. In particular, there is no limitation to 32-bit or 64-bit integers. This is an important property of the generator, as some static address computations may easily exceed a 64-bit limit. The integration of an arbitrary-precision arithmetic has been laborious, but prepares the generator for ultra-wide instruction sets, for instance in the area of cryptography.

The generator implements static evaluation by term rewriting. Therefore, an evaluation rule has been specified for each operation, as described in Section 7.3.4. This comprehensive set of evaluation rules can easily be extended, if operations are added to the primitive library. A constant subtree in the dataflow graph is evaluated from leaf nodes towards the root node. The evaluation is very fast, as term rewriting rules directly operate on integers using native operations.

7.4.2 Epsilon transformation

ViDL introduces the concept of epsilon logic, as described in Section 4.6. It is a unified concept, which can be used to express conditional as well as partial write accesses. Epsilon logic is an instance of a multi-value logic, which uses the states zero, one and epsilon. Although epsilon logic is well suited for specification, it can not directly be implemented in a simulator and in a hardware processor. Both rely on a binary representation of values, rather than ternary values.

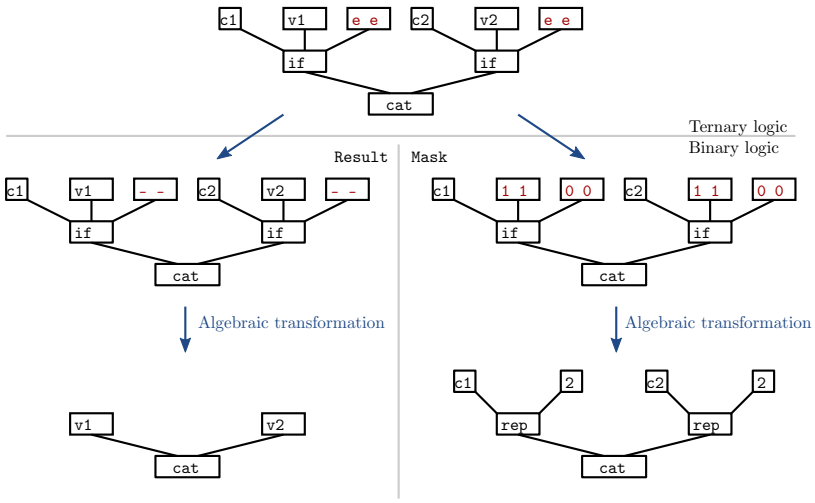


Figure 7.11: Transformation of epsilon logic into binary logic.

The front-end of the generator therefore contains an *epsilon transformation*, which transforms epsilon logic into binary logic. The transformation yields two binary expressions, one for the result and one for a write enable mask. These expressions are suited to control hardware registers and to implement lazy evaluation in the simulator (Section 7.5.4).

The idea of epsilon transformation is outlined in the following. Its implementation in the generator is actually very comprehensive and complex. It can therefore not be covered in-depth in this thesis. Figure 7.11 shows a simple example of epsilon transformation. It is used in the following to explain the transformation. For clearness, the illustration of the dataflow graph has been simplified.

The example shows the dataflow of a conditionally executed 2-way SIMD instruction, which operates on 2x2-bit vectors, i.e. 4-bit values. The parts of the result components are concatenated by the `cat` operation. Each component (`v1`, `v2`) of the result vector is written conditionally, depending on the condition `c1` resp. `c2`. For instance, if `c1` does not hold, the two most significant bits of the 4-bit result vector are set to `e`. As shown in the Figure, the ternary expression is transformed into a binary `Result` and a binary `Mask` expression. In the `Result` expression, `e` is replaced by "*don't care*" bits (`-`). If writing is disabled, these bits do not affect the processor state and can therefore be undefined. Don't care values are later exploited to simplify the dataflow graph. In the mask expression, each appearance of `e` is replaced by `0`, meaning "write-disable". In the example, the bits of `v1` and `v2` are replaced by ones, representing a write-enable. These `Result` and `Mask` expression may already be used to produce a processor implementation. However, to increase performance they are simplified by subsequent algebraic transformations. At the bottom of Figure 7.11, the

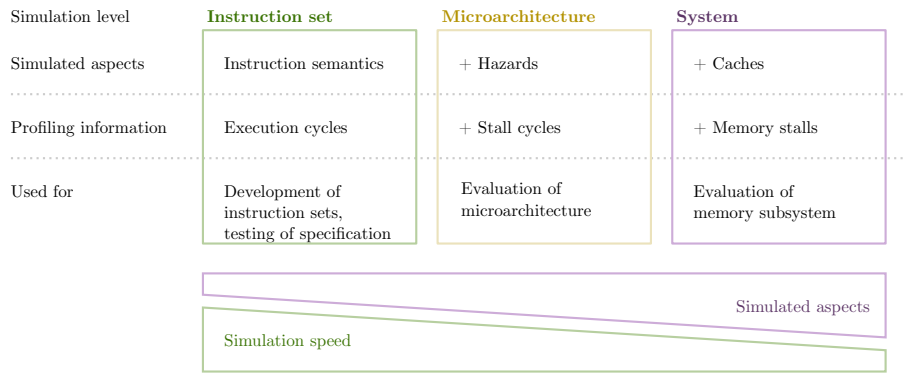


Figure 7.12: Levels of processor simulation.

optimized expressions are shown. The `if` operations in the result expression can be eliminated, as one alternative yields a "don't care" value. The transformation on the mask expression exploits, that the alternatives correspond to logical value of the conditions `c1` resp. `c2`. Each `if` operation can therefore be replaced by the condition. The final result and mask expressions contain only static operations on bits, which do not require any resources in a hardware implementation. `if`-operations, which result in costly multiplexers, have been eliminated entirely.

The basic idea of epsilon transformation is to split any subtree in the DFG that involves epsilon logic into a tree for its `Result` and a tree for its `Mask`. Both trees are then simplified by algebraic transformations, as demonstrated by the example. In addition, the implemented transformation also detects undefined applications of epsilon logic, which belong to the class of static semantic errors. For instance, the expression `add`$(5, \texttt{EPSILON})$ is not defined, as there is no meaningful interpretation of an addition on epsilon values. The generator also considers deeply nested epsilon expressions. Assume for instance that the ternary expression in Figure 7.11 is used in an add-operation. The generator reports a semantic error to the developer, since (depending on `c1` and `c2`), the expression may yield an epsilon result. However, if `c1` and `c2` are statically true, the generator does not complain, as the expression is always defined. Such an expression is of course simplified by subsequent algebraic transformations.

7.5 Methods for generating simulators

After discussing general translations and optimizations, this section focuses on methods for generating simulators. The methods are applied in the back-end of the simulator generator, which produces a simulator in terms of C-code. The code is ready for compilation, i.e. no additional line of C-Code has to be programmed by an instruction set developer.

Multiple kinds of simulators can be distinguished, which differ in their level of abstraction. This thesis distinguishes three such levels, which are illustrated

in Figure 7.12. On any level of simulation, the function of a program is accurately simulated, i.e. the result of simulation is the same, as program execution with respect to instruction semantics. The simulation speed and the amount of execution statistics however differs.

A simulation on *instruction set level* focuses on the semantics of instructions. The simulator that is generated from ViDL belongs to this class. It correctly executes any program of the specified instruction set. The simulator can be used to test the instruction set specification, as the functionality of instructions is accurately simulated. This includes delayed instructions, such as branches with delay slots. Remind that delays affect program semantics, as opposed to latencies (Section 4.3). A simulation on instruction set level can yield a wide range of profiling information. This includes precise statistics on the dynamic execution frequency of each instruction, dynamic data dependences, the flow of control and register accesses. Such information can be used to explore the instruction set for a given set of applications. In practice, a simulator in this level is also used to debug application software and compiler tools. On *microarchitectural level*, aspects of the microarchitecture are simulated in addition. This typically includes data-, resource- and control-hazards. Simulation on this level can yield statistics on stalls and utilization of bypasses. This information is used to optimize the microarchitectural processor implementation. However, such a simulation requires tracking of pipeline states and simulation of forwarding, which slows down simulation. On *system level*, components beyond the actual processor core are simulated. This includes the memory subsystem of the instruction memory and the data memory. A profiling also yields statistics on cache misses and hits. This level of simulation yields most information, but is also the slowest.

As described in Section 4.3, ViDL abstracts from all aspects of the microarchitecture. Accordingly, such aspects are not simulated. Instead, the simulator focuses on the instruction set, its functional behavior and its timing, by means of delays. In contrast, the processor generator implements methods to produce a microarchitectural processor implementation. However, these methods are not applied for the simulator, as they would slow down execution.

7.5.1 Structure of simulator

The simulation of an instruction consists of three steps, which are illustrated in Figure 7.13. First, an instruction word from memory is *decoded* (Section 7.5.3). The respective instruction is then *simulated* (Section 7.5.4), yielding a set of transactions. The *state transition* (Section 7.5.5) is finally performed by committing all transactions. Simulation and state transition have been decoupled to eliminate interferences and to enable delayed execution. These aspects are discussed later along with transactions. Before the phases of simulation are regarded in-depth, the representation of bit-strings is introduced in the next section. The representation of bit-strings is a key aspect of the generator, which touches all phases of simulation and has a major impact on simulation speed.

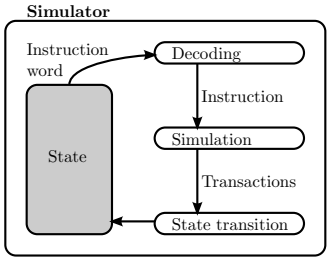


Figure 7.13: Layout of the simulator.

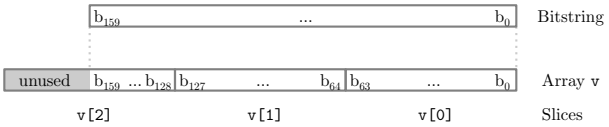


Figure 7.14: Representation of bit-strings in the simulator.

7.5.2 Bit-strings

In ViDL, any result and intermediate value is a bit-string. A bit-string may have any width in ViDL, i.e. the width is not limited by the language. A bit-string may have a length of one bit or even 3141 bits. The simulator needs to represent such bit-strings to store intermediate results. The representation needs to support ultra-wide bit-strings and should allow efficient simulation.

Some related approaches use *C integers* to represent intermediate values. However, such an approach does not scale. The width of bit-strings would be limited to 64 bit, which is the maximal size of integer variables in ANSI C. A naive implementation may represent a bit-string as an *array of bits*. This solution scales, but leads to very poor performance, as all operations have to operate on bits. For instance, a 64-bit addition will result in 64 computation steps. Besides, great parts of memory are wasted. A practical simulator may use an arbitrary-precision library, such as `libgmp`, to represent bit-strings. Such libraries consider the width of bit-strings to be dynamic, which increases flexibility. Unfortunately, this also leads to poor performance. It involves dynamic data structures and unbounded loops, which can not completely be unrolled by the compiler.

The generated simulator uses an approach similar to arbitrary-precision libraries, but exploits generation time information on bit-widths. A bit-string is represented as a static array of integers. An element of the array is called *slice* in the following. Figure 7.14 shows an example of a 160-bit wide bit-string and its representation as an array of three 64-bit slices. Slice number 2 stores only 32 significant bits of the bit-string. The remaining 32 bits are unused. Operations regard these bits to be undefined, which implies masking in some cases. Most operations however need not consider undefined bits. The size of the array corresponds to the width of bit-strings. Since the width of bit-strings is deter-

mined by type-inference at generation time, the size of the array is statically known. This enables an optimizing compiler to apply *scalarization*. As a result of scalarization, most slices can be mapped to registers of the host-processor, which leads to a very efficient execution. As an alternative, the simulator may directly represent each slice by a dedicated integer variable. However, shift instructions need random access to bits and therefore random access to slices. Such an access on integer variables can not be implemented efficiently. For this reason, the generator uses arrays of slices.

7.5.2.1 Operations on bit-strings

The generator breaks an operation on a bit-string down to integer-operations on slices. An `add` operation on 160-bit wide values is for instance broken down to the following ANSI C-code.

```
r[0]=v[0]+w[0]
r[1]=v[1]+w[1]+(r[0]<v[0] || r[0]<w[0])
r[2]=v[2]+w[2]+(r[1]<v[1] || r[1]<w[1])
```

The code contains only basic integer operations, which are likely mapped to corresponding host instructions. Since the arrays `r`, `w` and `v` are indexed statically, slices can be mapped to host registers after *scalarization*. The generator aims to produce straight-line C-code, as in the example. Such code does not include any control flow and can therefore efficiently be executed on deeply pipelined host processors. Thanks to static arrays, most loops can be unrolled by the generator. There are only few primitives, which utilize control structures of C.

7.5.2.2 Custom width of slices

The bit-width of slices can be defined by the user on invocation of the generator. It is not hard-coded in the generator and the code generators of primitives. This way, simulators can be generated that are tailored to the width of the host processor. For instance, a 32-bit simulator may be generated for a 32-bit host system and a 64-bit simulator for a 64-bit host. An n-bit simulator uses only n-bit integers and n-bit integer operations. Actually, the generator is not limited to these two bit-widths. It may as well produce a 128 bit[3] simulator or a 16 bit simulator.

7.5.3 Decoding

The task of the instruction-decoder is to determine the instruction of a given instruction word. The decoder may for example map the instruction word `1101001101010010` to the opcode "add-immediate".

Instruction-decoders are used in simulators as well as in compiler tools. They are for example part of disassemblers to decode instructions of executables and

[3] `gcc` defines a type `__128_t` on 64-bit hosts, however this is not a standard C-type of `stdint.h` to this day.

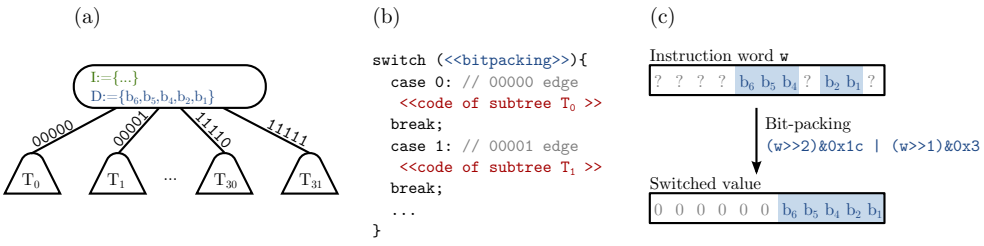


Figure 7.15: Example of decoder code generation for C.

object files. This section describes the generation of a decoder, in the context of a simulator. Nevertheless, the decoder can also be integrated in other tools, as it has a clear interface.

The input of the decoder generator is a decision tree, which is constructed by the front-end, as described in Section 7.1.4. The decision tree can be implemented in C in a straight forward way. The structure of the tree corresponds to the control flow of the generated code. Each node in the tree is mapped to a `switch` statement. Each child node corresponds to one case of that statement. Figure 7.15a shows a decision tree with subtrees T_0 to T_{31}. Figure 7.15b shows the respective C-code that implements the tree. Each case statement corresponds to one subtree. The code for each subtree is generated recursively. The case values correspond to the edge-annotations in the tree. The construction of the value `<<bitpacking>>` that is used for switching is illustrated in Figure 7.15c. Basically, the discrimination bits D of the instruction word are packed, such that they constitute a $|D|$ bit wide value. In the example, a 5-bit value is constructed, which yields an integral value in the range of 0 to 31. In C, bit-packing is implemented using only shift and logical operations. These operations are executed very fast on modern computers, compared to control flow. If a `switch` statement has only two cases (0 and 1), the generator replaces it by an `if` statement. Evaluation of the generated decoder has shown that approximately 150 million instructions are decoded per second on a 3 GHz host.

7.5.4 Implementing instruction semantics

For each instruction, the generator produces a sequence of C-Code, which implements its semantics. For instance, a branch instruction results in C-code, which computes the branch condition and the branch target. The code is generated from the *instruction DFG*, which represents the instruction's semantics. For the sake of simplicity, the DFG is assumed to be a tree in the following. The instruction DFG can be evaluated in a bottom-up pass. The code is therefore generated in the same order (bottom-up), as shown in Figure 7.16a. The code thereby obeys the data dependencies between DFG nodes, i.e. the code of a child A is executed before the code of a parent node B is executed. For a typical instruction this means, that addresses are computed first, followed by read operations, the actual computation and final issuing of a transaction.

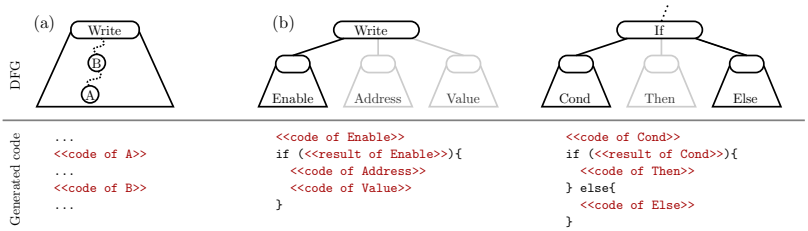


Figure 7.16: Simulation of instructions by evaluating the dataflow graph.

The simulation code of each nodes is produced by the respective generator, which is defined in the primitive library (Chapter 5). The generators are generic with respect to bit-widths and generate proper masking of slices. Wide primitives are broken down to simple integer arithmetic, as outlined in Section 7.5.2.1.

Lazy evaluation

A simple generator may produce C-code, which always computes the complete instruction DFG. Actually, this is not necessary in many cases. For instance, if the condition of a branch evaluates to false, the target address need not be computed. To exploit such optimization potential, the generator applies the idea of lazy evaluation, as it is found in some functional programming languages. To explain lazy evaluation, the DFGs in Figure 7.16b are regarded first. For a `write` node, the enable tree is evaluated first. If the write is enabled, the address tree and the value tree are evaluated. Otherwise, the evaluation is omitted, as the result is not used. Accordingly, the condition of the `if` node is evaluated first. Depending on the result, either the then-alternative or the else-alternative is evaluated. The evaluation of switch statements is similar, i.e. only one case is evaluated. In contrast to lazy evaluation, an eager evaluation would compute the result of all subtrees, even if it is not used.

The generated C-Code that obeys a lazy evaluation strategy is shown below each DFG. It is organized such, that the code of a subtree is only executed, if it needs to be evaluated. The code of the condition tree or enable tree is always executed. The code of the other trees instead is executed conditionally.

The application of lazy evaluation for dataflow graphs is sound, since the DFG is pure. Primitives do not have any side effects and their evaluation can therefore be omitted. If primitives would have side effects, lazy evaluation might break execution semantics. Note that lazy evaluation of write nodes relies on epsilon transformation (Section 7.4.2).

Lazy evaluation has a significant effect on simulation speed, as shown in Section 8.4.5.1. For instance, the target address of a conditional branch is only computed, if the branch is taken. Expensive computations of status flags are omitted, if they are not to be set (which is most likely). The result of a conditionally executed instruction is only computed, if the condition holds. On the other hand, the computation of the condition is omitted, if the instruction

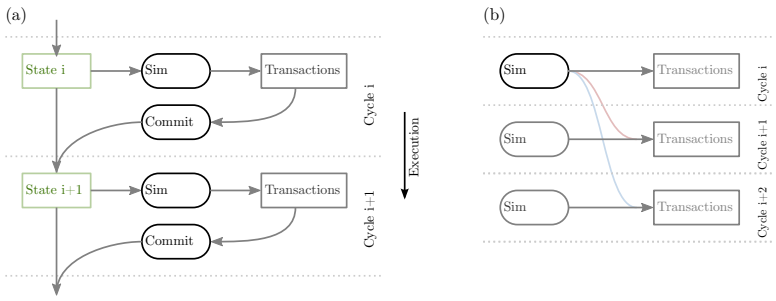


Figure 7.17: Decoupling of simulation and transition using transactions.

is executed unconditionally.

7.5.5 Transactions

The execution model of ViDL assumes an atomic change of processor state. This principle is resembled in the simulator using transactions. Transactions effectively decouple simulation from state transition. They are also used to implement delayed execution. The concept of transactions also prepares the simulation of parallel architectures, such as VLIW.

Basically, transactions defer state changes till the end of simulation. The principle of transactions is illustrated in Figure 7.17a. During simulation, each write access issues a transaction. The processor state is thereby not affected, i.e. the simulation code operates on the unmodified state i. After simulation, all transactions are committed, which corresponds to the actual transfer from state i to $i+1$. This transfer is atomic from the perspective of simulation.

As an alternative to transactions, double buffering may be used. Double buffering copies the state. One copy is used for read accesses and one for write accesses. In practice, the state is much larger than the number of transactions. Double buffering is therefore slow compared to the transaction base approach. The execution time for transactions is linear in the number of transactions and independent from state size. Its implementation in the generator is highly optimized. For instance, a static data structure is generated to hold transactions. This avoids expensive memory accesses, control code and allocation operations.

The generator also uses the transaction mechanism to implement delayed write accesses. A write access can be delayed, by issuing the respective transaction to a future cycle. In Figure 7.17b, the simulation code of cycle i issued a transaction to the current cycle, to cycle $i+1$ (red), and one transaction to cycle $i+2$ (blue). These transactions are then committed with no delay, a delay of one and a delay of two cycles. The data structures that store transactions are organized in a ring, where the size of the ring corresponds to the maximal delay that is specified in ViDL. A delayed transaction is issued and processed in constant time. The actual delay and the number of outstanding accesses do not affect execution time.

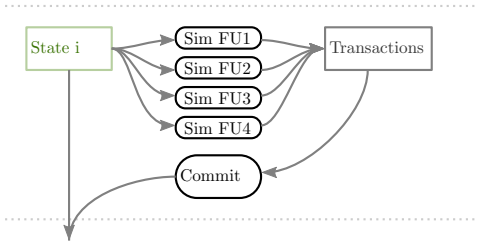


Figure 7.18: Simulation of parallel processors using transactions.

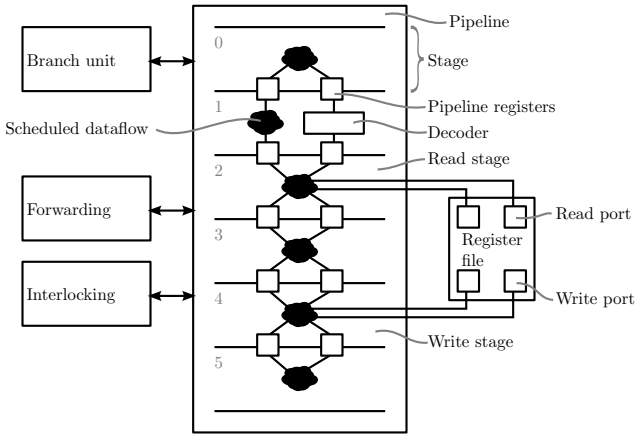


Figure 7.19: Components of the generated microarchitectural processor implementation.

The concept of transactions prepares the simulator for simulation of parallel functional units. For instance, in a 4-way VLIW processor, all 4 functional units can be simulated sequentially, as show in Figure 7.18. After simulation, the collected transactions are committed. The simulation of parallel instructions does thereby not interfere. The approach scales well, since the overall execution time is linear in the number of functional units. Using multiple transaction buffers, functional units can even be simulated concurrently, which may increase simulation speed on multi-core systems.

7.6 Methods for generating processors

This section describes the generation of a microarchitectural processor implementation. All methods that are described in this section are implemented in the processor generator, which is evaluated in Section 8.5. An overview of a generated processor structure is shown in Figure 7.19. The Figure is just meant to give a basic impression of the structure of one specific processor. The processor consists of a pipeline and a register file. The pipeline has multiple stages,

which are divided by pipeline registers. An instruction is fetched in stage 0, decoded in stage 1, traverses the entire pipeline and is finally retired in stage 5. The execution of instructions in the pipeline is controlled by the branch unit and interlocking. Parts of the dataflow graph are represented by black clouds, which are distributed among the entire pipeline.

The figure is referred in the following, to introduce the components of the processor generator. The processor's instruction decoder is produced by a *decoder generator*, which is similar to the decoder generator of the simulator (Section 7.5.3). The next phases are highly dependent, i.e. one phase relies on the result of the previous phase. The first phase called *port assignment* (Section 7.6.1) generates read and write ports for each register file and each data memory. The example in Figure 7.19 shows a register file with two read ports and two write ports. Before the pipeline is constructed, some primitives (e.g. multiplier) may be pipelined by the *operation pipelining* phase (Section 7.6.2). This phase affects the latency of operations and must therefore be executed before any other pipeline related phase. The read stage and the write stage of each storage are determined in the subsequent *port scheduling* phase (Section 7.6.4). The resulting stages are connected to the read and write ports that have been derived in the previous phase. In the example (Figure 7.19), the register file is read in stage 2 and written in stage 4. Depending on the location of read and write ports, the remaining nodes of the DFG are scheduled by the *operation scheduler* (Section 7.6.5). The dataflow that is assigned to stages is indicated by black clouds in the example. Once all nodes of the DFG are scheduled, *pipeline registers* can be inserted (Section 7.6.6). The insertion of pipeline registers depends on the previous operation scheduling. As shown in the example, pipeline registers are placed on the boundaries between stages, which effectively interrupt dataflow between stages. To eliminate resulting data-hazards, a *forwarding circuit* (Section 7.6.7) is generated for each register, unless such hazards can be excluded. The generation involves sophisticated analysis of the dataflow. Forwarding is always applied, unless it would introduce further critical paths. The forwarding circuit must be inserted *after* pipeline registers have been inserted. Otherwise, the DFG would become a cyclic graph. If a data-hazard can not be resolved by forwarding, *interlocking* (Section 7.6.8) is applied. The generation of interlocking is based on analysis information from the previous forwarding phase. The processor generator aims to construct pipeline control such, that the number of pipeline stalls is minimized. The final phase of the transformation constructs a *branch unit* (Section 7.6.9). The branch unit applies speculative execution and instruction canceling in case of misprediction. The generation involves analysis of dataflow and optimizations to simplify control logic. It depends on results from the forwarding phase.

7.6.1 Register port allocation

A processor typically includes one or more register files. Each register file has a set of read ports and a set of write ports, as described in Section 2.7.1. A register file can be accesses concurrently through these ports. One read access

or write access can be performed through each port at a time.

In the DFG, read and write accesses to a register file are represented by read and write nodes. To construct an efficient processor implementation, these access nodes must be assigned statically to ports. The assignment has to consider two aspects. First, the assignment must not introduce any *resource-conflicts* and second, the number of *register ports* should be minimized. Register ports are an expensive resource in terms of chip area and power consumption. A minimal number of read-ports and write-ports that does not lead to resource conflicts is therefore desirable.

A resource conflict exists, if two access nodes that are assigned to the same port are activated concurrently. For instance, assume a 3-address instruction that reads from registers via read nodes X and Y. As these nodes are activated concurrently by the instruction, they must be assigned to different read ports. The register file must therefore include at least two read ports.

What makes port assignment hard is the fact, that access nodes in the DFG have likely been merged by previous phases. An access node is therefore associated with a set of instructions rather than a single instruction. For instance, read nodes of two instructions are merged, if they use the same addressing mode. In practice, the DFG contains about ten read nodes, which are associated with numerous instructions. Merging of write nodes is less common, but actually appears. For instance, the write nodes for auto-increment of load and store instructions are typically merged. Figure 7.20a shows a simplified DFG of 5 read nodes R1 to R5, and dataflow that belongs to instructions A through D. The read node R1 is therefore a part of instructions A and B, i.e. R1 is activated by A and B. The color of a read node corresponds to the assigned port. The computation of the assignment is explained in the next section.

7.6.1.1 Assignment algorithm

This section describes the assignment algorithm that is implemented in the processor generator. To simplify explanation, the assignment of read nodes to read ports is regarded for one register file.

Basically, the assignment problem is solved by coloring a graph $G = (V, E)$. Each node $v \in V$ in the conflict graph corresponds to one read node. An edge between two nodes expresses, that the respective accesses may be activated concurrently, i.e. they must not be assigned to the same port. In detail, two nodes are connected by an edge, if the sets of activating instructions are not disjoint.

$$E = \{(u, v) | \text{active}(u) \cap \text{active}(v) \neq \emptyset\}$$

The function $\text{active}(x) \subseteq \text{Instructions}$ yields the set of activating instructions for a node x.

A color corresponds to a read port. A coloring of the graph thereby assigns read ports to read nodes, where the chromatic number of the conflict graph corresponds to the number of read ports. To minimize the number of read ports, the graph must be colored conflict free using a minimal number of colors

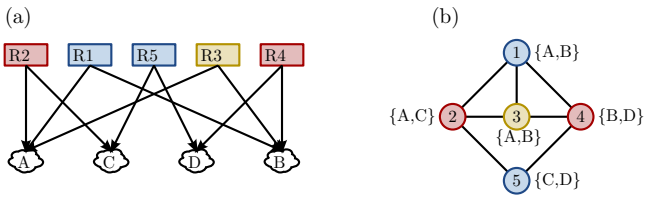


Figure 7.20: Conflict graph for port assignment.

Figure 7.20b shows the conflict graph of the adjoining DFG. The set of nodes $V = \{1..5\}$ corresponds to the read nodes R1 to R5. Each node in the conflict graph is annotated with the set active(x) of activating instructions. The read access of node 1 is for instance activated by instructions A and B. Nodes 1 and 2 are connected by a conflict edge, as they are both activated by instruction A. As shown, the conflict graph can be colored using 3 colors. It can not be colored with fewer colors, as nodes 1, 2, and 3 constitute a clique. The chromatic number of the graph is 3, i.e. the minimum number of required read-ports to avoid resource conflicts is 3. In practice, the constructed conflict graph is a sparse graph. It contains only few edges and is efficiently colored by the processor generator.

7.6.2 Operation pipelining

To increase the clock frequency of generated processors, operations may be pipelined by the generator. For instance, the multiplier may be pipelined, such that it is distributed among multiple stages. However, pipelining of operations typically increases instruction latencies, chip area and power consumption of the processor. It is therefore only applied if necessary. The decision if an operation is pipelined depends on its estimated delay and the user supplied target frequency, which is introduced in the next section.

Operation pipelining is implemented using the term rewriting system and timing sensitive rewrite rules. The latter are defined using rule sets, which have been described in section 7.3.5. The target frequency is therefore divided into different ranges. For each range, one rule set is defined. The rules in this set then translate non-pipelined operations into their pipelined counterparts.

7.6.3 Timing

So far, we have a simple DFG, which defines the mapping from the current processor state to the next processor state. This DFG could immediately be used to generate a non-pipelined processor. To generate a pipeline processor, the generator has to consider two kinds of timing, namely the *timing of operations* in the DFG and the targeted timing of the processor. The latter is supplied in terms of a *target frequency* f_{target} by the user. The timing of each operation is defined in a so called *delay library*. The library defines propagation delays

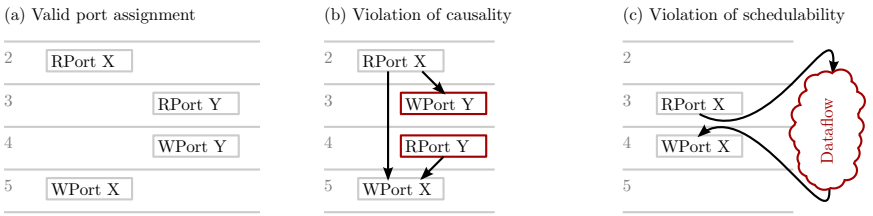


Figure 7.21: Examples of valid and invalid port schedules.

and latencies for all operations. The propagation delay depends on the chip technology that is targeted during synthesis. For this reason, multiple delay libraries may be specified, one for each technology.

Pipeline related decisions of the generator are guided by the timing of operations and by the user defined target frequency f_{target}. The generator tries to organize the pipeline such, that the critical path does not exceed the timing constraint f_{target}^{-1}. On the other hand, unnecessary pipelining is avoided to save resources. The target frequency is a significant parameter, which is used in practice to guide the generation of processors. It has a major impact on the processors frequency (as determined by synthesis), chip area, power consumption and instruction latencies. Using f_{target}, the developer can generate very different processor implementations from the same instruction set specification. The relation between f_{target} and physical characteristics of the processor is evaluated in Section 8.5.3. Its impact on instruction latencies is discussed in Section 8.5.9.

The remaining sections describe methods to construct an instruction pipeline. Most methods consider the target frequency and the timing of operations in some respect.

7.6.4 Port scheduling

The port scheduling phase determines the read stage and the write-back stage of each storage, by assigning read ports and write ports to stages. For instance, the read ports of a register `X` may be assigned to stage 2 of the pipeline and the write ports may be assigned to stage 5, as shown in Figure 7.21a. In this case, stage 2 is called the read stage and stage 5 is called the write-back stage of `X`. Note, that distinct stages may be assigned for each storage, as demonstrated in the example for `X` and `Y`.

For port scheduling the generator considers the structure of the DFG, the propagation delays of its nodes and the targeted frequency of the final processor. The resulting port assignment must be valid and should be optimized. The optimization criteria are discussed later.

A valid schedule must satisfy two constraints, namely *causality* and *schedulability*. The causality constraint demands that the write stage of a storage Y must not precede the read stage of Y. Otherwise, a processor state i may depend on a state $i+1$, which violates causality. In Figure 7.21b, causality is violated, as `Y` is written before it is read. For data dependent read-write accesses (see

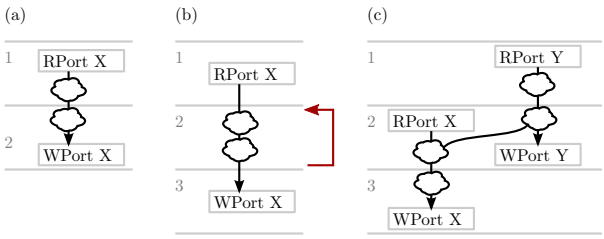


Figure 7.22: Example of valid port schedules of different quality.

storage `x`) causality always holds. However, if there is no such dependence, causality may be violated. The causality constraint is therefore enforced by the port scheduling of the ViDL generator.

The second constraint regards the schedulability of the entire dataflow graph. The constraint demands that the dataflow graph can be scheduled among pipeline stages, as described in Section 7.6.5. If data dependent ports are scheduled too tight, as in Figure 7.21c, the intervening dataflow may not be schedulable without violating timing constraints. The port scheduler therefore considers the target frequency f_{target} and the timing of DFG nodes.

Any port schedule that satisfies the previously mentioned constraints can be used to implement a processor. However, the implementation may be inefficient. In particular, the pipeline of the processor and write latencies may be unnecessarily long. Figure 7.22 shows two valid port assignments for the storage `X`. The first assignment (a) requires two pipeline stages, whereas the second (b) uses three stages and requires forwarding (red).

To yield an efficient processor implementation, the port scheduler aims to minimize the write latency and the pipeline length. The write latency is defined as the difference between the write and read stage of a storage. A low latency reduces data-hazards and simplifies the forwarding circuit. A long latency results in many pipeline bypasses. The implemented port scheduler also aims to minimize the pipeline length. Figure 7.22c shows a more complex dataflow graph and a port assignment, which minimizes write latencies as well as the pipeline length.

7.6.4.1 Algorithm

The scheduling of ports can be modeled as a linear integral optimization problem. For each storage one variable is introduced for the read ports and one variable for the write ports. The variables represent the stage of the ports. For n storages this makes $2n$ variables. The scheduling constraints are expressed by inequalities on the variables. The objective function is given by the distance between read and write stages, which is to be minimized. The solution of this problem corresponds to a valid assignment of stages to each port.

7.6.4.2 Bypass-relaxing

If the distance between read and write stages is larger than one, a forwarding circuit is to be inserted in a later phase of generation (Section 7.6.7). The bypasses of the forwarding circuit may introduce further critical paths. These critical paths of bypasses are relaxed at the expense of an additional propagation delay on the datapath. The delay in turn affects the port scheduling and may increase the distance between ports.

The port scheduler therefore iteratively schedules ports and relaxes bypasses. The number of forwarding circuits thereby rises monotonic. As the number of forwarding circuits is limited by the number of storages, the iterative computation is guaranteed to converge. In practice, the ViDL generator likely reaches a fixpoint after two iterations.

7.6.5 Operation scheduling

After registers ports have been scheduled, the remaining nodes of the DFG are assigned to pipeline stages. The scheduler is similar to the port scheduler. It considers the user supplied target frequency f_{target}, the port schedule, the DFG, and the estimated propagation delay of its nodes.

7.6.5.1 Modeling

The scheduling constraints on DFG nodes can be modeled by a system of equations. The schedule of a node x is characterized by two values: The *stage* s_x of the node and the *arrival time* a_x within the stage. The arrival time a_x is the estimated signal delay from pipeline registers to x's inputs. The *propagation delay* that is contributed by node x is denoted d_x in the following. It must lie in the range of the *period* p, which is the inverse of the user supplied clock frequency.

$$p = f_{\text{target}}^{-1}$$
$$0 \leq d_a \leq p$$

If this property does not hold, the DFG is not schedulable and the generator reports an error. The user then has to reduce the target frequency f_{target}.

The *finish time* at x's outputs is given by the node's arrival time and propagation delay

$$f_x = a_x + d_x.$$

For instance, a node x with a propagation delay of $d_x = 800\,\text{ps}$ (picoseconds) may be scheduled in stage $s_x = 2$ with an arrival time of $a_x = 150\,\text{ps}$, which results in a finish time of $f_x = 950\,\text{ps}$. For read and write nodes, the stage is already fixed by previous port scheduling. These variables can therefore be considered constant.

$$s_x = s_{\text{port}(x)}$$

7.6.5.2 Scheduling constraints

The scheduling constraints are modeled by inequalities on the variables that have previously been introduced. The timing of a node must lie within the user defined period p. Otherwise, the generated processor may not yield the targeted clock frequency.

$$0 \leq a_x \wedge f_x \leq p$$

For two data dependent nodes $x \rightarrow y$, the result of x must be available before it is expected by y. This constraint holds, if x is located in an earlier stage then y. In this case, both nodes are separated by pipeline registers and there is no direct dataflow between both nodes. If instead both nodes are located in the same stage, the finish time of x must be smaller than the arrival time of y.

$$s_x < s_y \vee (s_x = s_y \wedge f_x \leq a_y)$$

Note that this equation implies $s_x \leq s_y$, i.e. the dataflow between x and y must obey the dataflow direction of the pipeline. If s_x would be greater then s_y, data would flow from later stages to earlier stages.

The processor generator solves the system of equations by a two pass algorithm in linear time. For the ARM instruction set, stage numbers are assigned to approximately 2.000 nodes in less then a minute. The arrival and finish times are discarded after scheduling, as they are not needed for any subsequent transformations.

7.6.6 Pipeline registers

The previous assignment of DFG nodes to stages implies the location of pipeline registers. If two consecutive nodes in the DFG are assigned to different stages, pipeline registers have to be inserted on their connecting edge. For two nodes $x \rightarrow y$ exactly $s_y - s_x$ pipeline registers have to be inserted between x and y.

A pipeline register interrupts the dataflow between nodes. In the dataflow graph, a pipeline register is represented by two associated nodes. A write node in stage s, which receives a value and a read node in stage $s + 1$, which yields a value. Figure 7.23 show an example for the insertion of two pipeline registers between nodes X and Y. Each pipeline register consists of the pair `PW` and `PR`. These nodes are associated, but not connected in terms of the DFG. After insertion, nodes of different stages are not connected by a path in the DFG. The dataflow graph is thereby divided into at least n connected components, where n is the number of pipeline stages.

7.6.7 Forwarding circuit

The pipelined DFG that has been constructed so far can be used to generate a pipeline processor implementation. However, the implementation likely includes unresolved data-hazards, which appear as write delays. These hazards

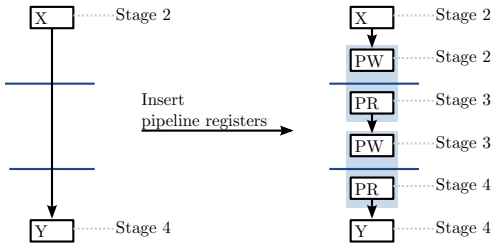


Figure 7.23: Insertion of pipeline registers.

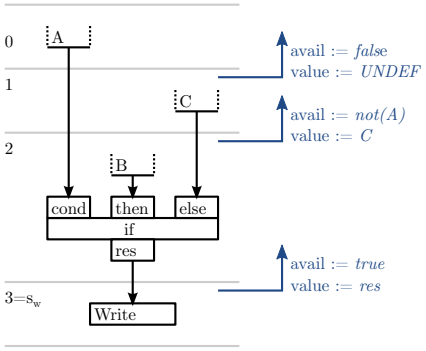


Figure 7.24: Example of early result forwarding.

must be resolved, to retain execution semantics of instructions. Hazards are primarily resolved by forwarding as described in this section. If forwarding can not be applied due to latencies, interlocking is applied to resolve the hazard (Section 7.6.8).

The principle of forwarding has been explained in Section 2.7.5. This section describes the construction method for a forwarding circuit, as it is implemented in the generator. To give an impression of the construction task, a small forwarding example is discussed first. Using this example, the actual construction method is explained.

For a storage with read stage s_r and write stage s_w, bypasses must be inserted from all stages $s, s_r + 1 < s \leq s_w$. Actually, a bypass must be inserted for each stage s, each write port and each read port. The bypasses are then merged by a generated multiplexer in stage $s_r + 1$.

What makes forwarding hard, is the fact that the generator has to decide, in which stage a result is fed into the forwarding circuit and under which condition. A result can only be bypassed from a stage s, if it is already available in that stage. Figure 7.24 shows a highly simplified example of forwarding. The values A, B and C are fed into a condition node (if). Its result is written back in stage $S_w = 3$. The positions of the nodes A, B and C are assumed to be fixed, as shown in the Figure. Due to the position of B, the if-node must be scheduled in stage 2. The final result can therefore only be forwarded from the beginning of

stage 3 ("value:=res" in the Figure). The result is always available in this stage, which is expressed by "avail:=true".

In stage 2, the final result is not computed yet. However, if A does not hold, the result is equal to C, which is already available at the beginning of stage 2. The value of the bypass is therefore set to C and the availability to the expression *not(A)*. In stage 1, the result is not known in any case. The availability is therefore set to *false* and the value to *UNDEF*. A bypass from this stage is therefore omitted.

The structure shown in the example is typically produced by the generator's port and hardware sharing transformations. The term A may for instance distinguish between two instructions, which yield the results B and C.

7.6.7.1 Construction method

The previous example was intended to give a rough idea on construction of forwarding. This section describes the method that is implemented in the generator to systematically construct the forwarding circuit.

In the following, the value that is written-back is regarded as a term t. For the example this is

$$t = \text{if}(A, B, C)$$

The algorithm derives the "*value*" and the "*defined*" expression for each stage s separately. Therefore, the write back node is conceptually scheduled to stage s. This typically leads to a violation of scheduling constraints.

The violation is resolved by a so called *thin-out* phase. This phase removes all subterms from t that contribute to the violation. For the example and stage $s = 2$, node B is removed, which results in the term

$$t_{\text{thin}} = \text{if}(A, \text{UNDEF}, C)$$

The term t_{thin} thereby becomes partially defined. If A holds, the result of the term is undefined.

A subsequent *is-defined analysis* determines when the term is defined. For the example, the analysis yields the expression

$$t_{\text{def}} = not(A)$$

If this expression holds, the term t_{thin} is defined, otherwise it is not. The expression t_{def} is then used to control interlocking, as described in the next section.

In a third *reduction* phase, the term t_{thin} is simplified, yielding a term t_{red}. Basically, all parts of the term that contribute only to an undefined result can be removed. This likely enables the simplification of other operations in the term. In the example, the "if" operation can be eliminated, as only one alternative yields a defined value. This also eliminates the subterm A, which evaluates the condition. The resulting simplified term is

$$t_{\text{red}} = C$$

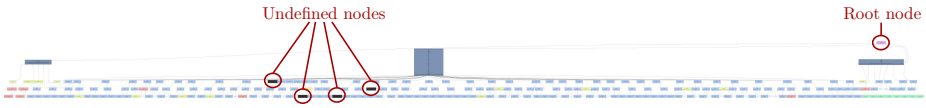


Figure 7.25: Expression tree of a partially defined result term as used to derive the forwarding of a 6-stage MIPS processor.

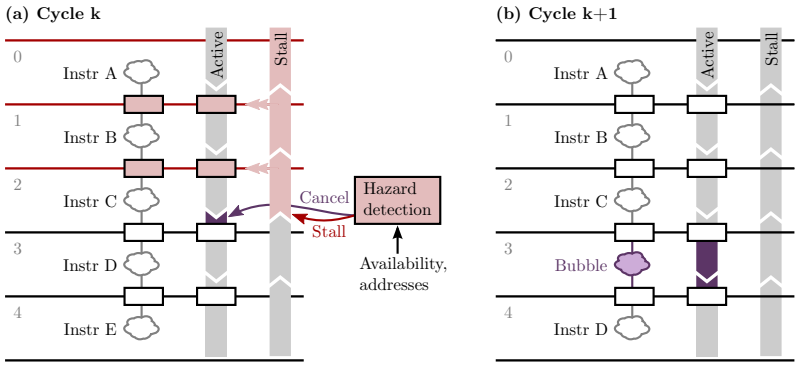


Figure 7.26: Scheme of constructed interlocking.

This simplified term $t_{\text{simp}} = C$ is finally fed into the forwarding circuit, as shown in stage 2 of Figure 7.24. Timing constraints of the forwarding circuit are obeyed due to previous thin-out phase. If t_{def} holds, the forwarded value t_{simp} is equal to the value t that is finally written back to the register. If t_{def} does not hold, the undefined forwarded value is not used due to interlocking. Interlocking is discussed in the next section.

This section gave only a very simple example to demonstrate the forwarding-insertion method. In practice, the generator processes terms that consist of hundreds of nodes, including nested conditions, arithmetic operations and switch statements. Figure 7.25 gives an impression of such a term. It has been exported by the processor generator during generation of a 6 stage MIPS processor. The term represents the partial result t_{thin} of the GPR registers at the end of the execute stage (EX). The black nodes are undefined, as they would lead to a violation of scheduling constraints. One of these nodes is for instance a read access on data memory, which is scheduled in the next stage (MA).

7.6.8 Interlocking

The previous section explained how availability information is computed. This information is now used to control the pipeline, as shown in Figure 7.26a. The availability information from all stages is fed into a hazard detection circuit, along with the addresses of forwarded values. If a forwarded value is accessed, but not yet available, this circuit stalls the pipeline and inserts a bubble. Therefore, the processor generator creates two signal paths called “active” and “stall”,

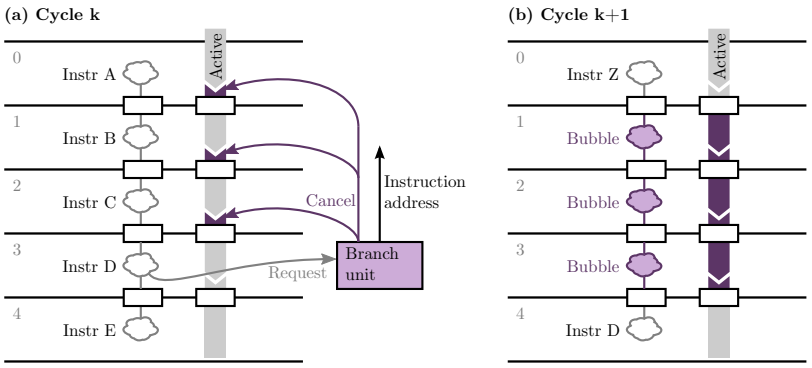


Figure 7.27: Example of a branch and its processing by the branch unit.

which range through all stages. The *stall* path propagates stall information upwards, i.e. from stages i to stages $j, j < i$. Note, that this path is not interrupted by any pipeline registers. In case of a data hazard, the detection circuit triggers a stall. In the example, the stall is triggered in stage 2, which is then propagated to stage 1 and 0. The stall path disables the pipeline registers between stages 0/1 and 1/2 (drawn red in the Figure). Thereby, instructions B and C remain in stages 1 and 2. In addition, the instruction fetch logic (IF) is controlled such, that instruction A remains in stage 0.

To insert a bubble in stage 3, the *active-path* is used. The hazard detection circuit triggers an instruction cancel at the end of stage 2. Note, that this does not affect any instruction, until the next clock cycle. On the next clock edge, the cancelation is passed to stage 3, as shown in Figure 7.26b. This stage is thereby deactivated in stage $k + 1$, i.e. it is regarded to contain a bubble.

Actually, the explanations given in this section are highly simplified. The construction method that is implemented in the processor generator has to consider additional aspects of a pipeline of arbitrary length. For instance, hazard detection also considers write enable signals and has to prioritize availability information according to originating stages. A stall signal need not only disable pipeline registers, but also has to disable registered memories. To optimize the generated processor, the active- and stall-paths are only generated in part. The active-path likely starts in the execute stage and logic to interconnect hazard detection is omitted in most stages. This depends on the specified instruction set and the generated microarchitecture. The hazard detection circuit is simplified, by eliminating address comparisons for single registers and static addressing.

7.6.9 Branch prediction

The processor generator constructs a branch unit based on the specified instruction set and the derived pipeline structure. Its complexity basically depends on the number of branch instructions that have different timings. Currently, the generated branch unit implements a very simple “not-taken” prediction scheme.

However, the generation method can be extended to yield other branch predictors. This has not been subject to the evaluation of this thesis.

The branch unit basically has two tasks: To supply the (predicted) *next instruction address* and second, to *cancel* speculative instructions in case of misprediction. For the "not-taken" scheme, the branch unit predicts the next instruction address to be the successor of the current address.

To change the control flow, the branch unit receives branch requests from branch instructions. The branch requests may typically be issued from multiple pipeline stages. Figure 7.27a shows the effects of a branch instruction (Instr D). The branch instruction issues a branch request from stage 3, which is processed by the branch unit. As the prediction scheme is "not-taken", the branch unit cancels instructions A to C, which have been executed speculatively. The cancelation takes effect from cycle $k + 1$ on, as shown in Figure 7.27b.

A branch request consists of two values, the branch's *target* address and a *taken* flag, which triggers the request. Using such a pair of signals, multiple stages are connected to the branch unit. Each stage that is connected may trigger a branch. Based on the instruction set and propagation delays, the generator determines, which stages are to be connected. For instance, an "unconditional relative" branch is likely issued from stage 1, a "register" branch from stage 2 and a "register+offset" branch from stage 3. The generator may therefore connect stages 1 to 3 to the branch unit. The branch unit then prioritizes requests with respect to their originating stage. The construction of the *target* and *taken* signal uses methods from forwarding and interlocking construction.

Since the branch unit may cancel instructions, additional constraints must be considered by the generator. In general, a branch request from stage i may cancel instructions in stages 0 through $i - 1$. The instructions in these stages must therefore be cancelable. An instruction is cancelable in a specific stage, if all its effects on the processor state can be eliminated. That is the case, if the instruction has not committed any changes by register write-back. As a result, write-back stages must not precede the latest stage that may trigger a branch. This constraint is already considered during port scheduling 7.6.4.

Chapter 8

Evaluation

Both generators have been developed beyond the state of prototypes. They have been tested and can be considered quite stable and correct. Extensive amounts of simulators and processors have been generated using the system. Almost all aspects of the instruction sets are properly implemented in the generated products, including ultra-wide operations, conditional execution and partial write accesses. Exceptions are discussed in Section 8.2.5. Both, simulators and processors are generated *entirely and solely* from the ViDL specification. There is no patching or fine-tuning involved. No single line of C or VHDL code has been added to the generated code.

8.1 Evaluation process

The entire evaluation process is visualized in Figure 8.1. To show the fitness of ViDL, 13 instruction sets and ISA variations have been specified (Section 8.2). This includes ARM, MIPS, Power, OISC, SRC, and the DNACore instruction set extension of MIPS. The latter is discussed in the context of a case study on instruction set extension (Section 8.6). The specification of SRC is covered in Section 3.6.6, where it is compared to a respective Lisa specification. Since the evaluation is quite comprehensive, it has been automated using a set of evaluation scripts. For instance, a build mechanism repeatedly invokes the simulator and processor generator, yielding a set of simulators and a set of processors, as shown in the figure. The generators have been tuned with respect to generation time, such that the entire generation process takes only a short time (Section 8.3).

The generated simulators are evaluated (Section 8.4), by executing binary programs of the respective instruction set. The programs have been written in assembler and include micro-benchmarks, as well as algorithmic kernels. The processor implementations are generated from the very same ViDL specifications. For each instruction set specification, multiple processors are generated. The processors differ in their microarchitectural implementation, i.e. pipeline

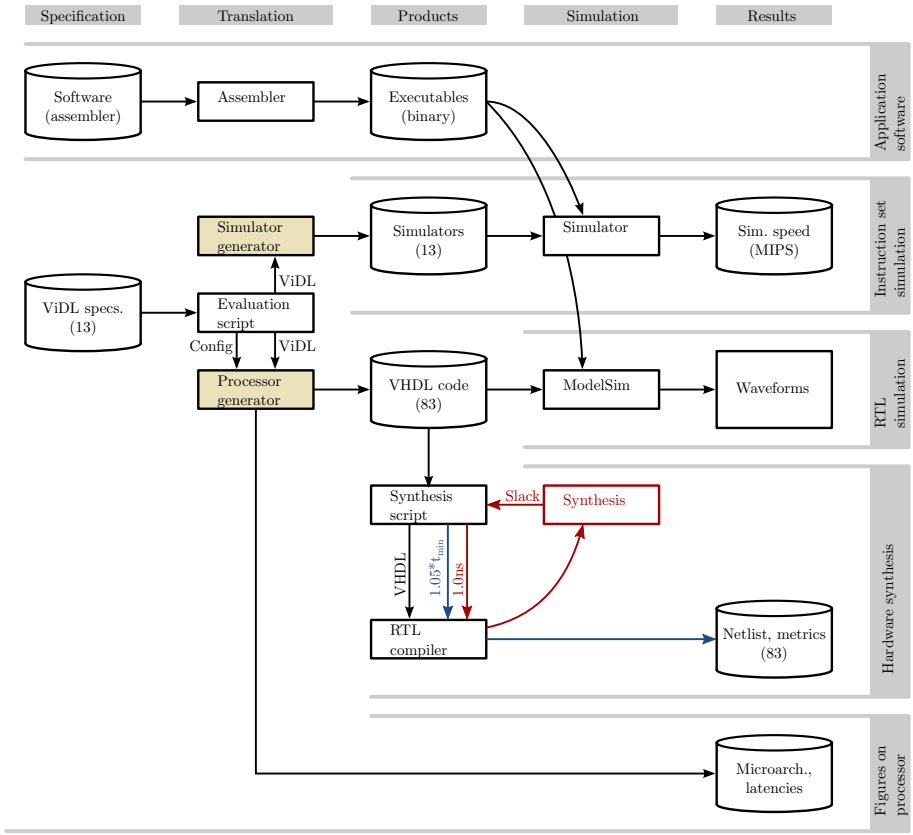


Figure 8.1: Overview of the automated evaluation process.

depth, forwarding and interlocking. For each processor, the build mechanism invokes the processor generator with a combination of a configuration and ViDL specification. The result is a set of 83 processors in terms of VHDL code and figures on their microarchitectural implementation.

The correctness of processors has been tested on register transfer level (RTL) using ModelSim (Section 8.5.10). Therefore, binary test programs have been executed on the simulated processor. Test cases cover exceptional instructions and aim to stress the pipeline of the processor. Some programs are self checking and some have been validated by inspecting the waveforms of significant processor signals. Apart from testing, simulation also yields information on the processor's dynamic behavior, in terms of stalls, branch penalties and the resulting CPI[1].

As shown in the figure, each processor has been synthesized, to estimate its clock frequency, chip area, and power consumption (Section 8.5.2). This process has been automated by a script, which invokes the RTL compiler with different timing constraints. In a first pass (red), synthesis is constrained very hard with 1 ns, to get the fastest, but feasible timing from synthesis tools. In a second pass (blue), this timing (with an addition of 5%) has been used to estimate the physical processor characteristics. The netlist from synthesis has also been tested my means of a gate-level simulation in ModelSim.

8.2 ViDL

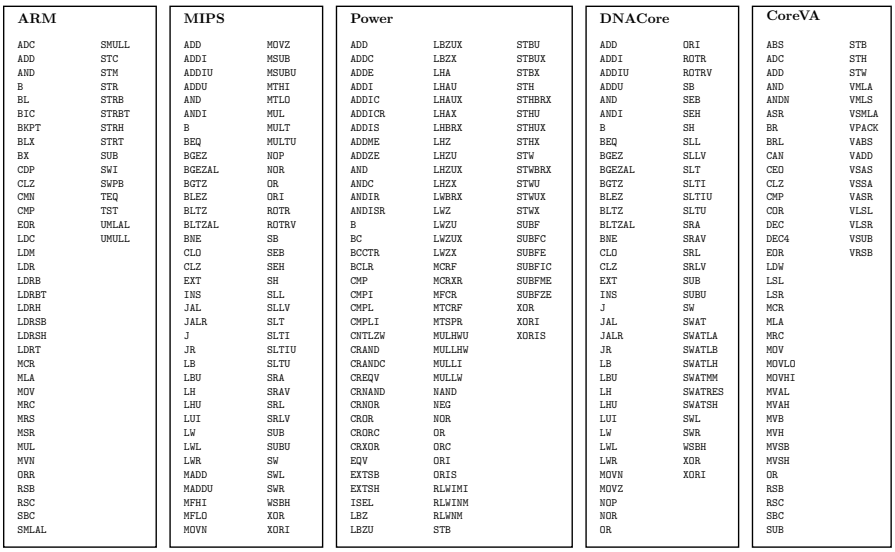
This section evaluates the application of ViDL in practice. Language properties, such as maintainability, simplicity and reliability have already been discussed along with the description of ViDL (Chapter 4). To demonstrate the power of ViDL, real world instruction sets have been specified (Section 8.2.1). The specifications are compact and have been defined in a comparably short time (Section 8.2.2). Some instruction sets have been defined by inexperienced users, which highlights the simplicity of ViDL (Section 8.2.3). The formalized instruction set of ARM has been explored with respect to bit-widths of computations (Section 8.2.4).

8.2.1 Real world instruction sets

ViDL is suited to precisely specify real world instruction sets. This property guarantees the applicability of ViDL in serious projects, apart from idealized academic instruction sets. For evaluation, ViDL has been used to define four general purpose instruction sets, namely ARM, MIPS, Power and CoreVA. The amount of instructions that have been specified is shown in Figure 8.2. It includes almost all arithmetic, logic, memory and branch instructions. Restrictions are discussed in Section 8.2.5.

As you can see in the Figure, the DNACore instruction set is nearly identical to the MIPS instruction set, except that multiplications have been omitted

[1] cycles per instruction



ARM

ADC	SMULL
ADD	STC
AND	STM
B	STR
BL	STRB
BIC	STRBT
BKPT	STRH
BLX	STRT
BX	SUB
CDP	SWI
CLZ	SWPB
CMN	TEQ
CMP	TST
EOR	UMLAL
LDC	UMULL
LDM	
LDR	
LDRB	
LDRBT	
LDRH	
LDRSB	
LDRSH	
LDRT	
MCR	
MLA	
MOV	
MRC	
MRS	
MSR	
MUL	
MVN	
ORR	
RSB	
RSC	
SBC	
SMLAL	

MIPS

ADD	MOVZ
ADDI	MSUB
ADDIU	MSUBU
ADDU	MTHI
AND	MTLO
ANDI	MUL
B	MULT
BEQ	MULTU
BGEZ	NOP
BGEZAL	NOR
BGTZ	OR
BLEZ	ORI
BLTZ	ROTR
BLTZAL	ROTRV
BNE	SB
CLO	SEB
CLZ	SEH
EXT	SH
INS	SLL
JAL	SLLV
JALR	SLT
J	SLTI
JR	SLTIU
LB	SLTU
LBU	SRA
LH	SRAV
LHU	SRL
LUI	SRLV
LW	SUB
LWL	SUBU
LWR	SW
MADD	SWL
MADDU	SWR
MFHI	WSBH
MFLO	XOR
MOVN	XORI

Power

ADD	LBZUX	STBU
ADDC	LBZX	STBUX
ADDE	LHA	STBX
ADDI	LHAU	STH
ADDIC	LHAUX	STHBRX
ADDICR	LHAX	STHU
ADDIS	LHBRX	STHUX
ADDME	LHZ	STHX
ADDZE	LHZU	STW
AND	LHZUX	STWBRX
ANDC	LHZX	STWU
ANDIR	LWBRX	STWUX
ANDISR	LWZ	STWX
B	LWZU	SUBF
BC	LWZUX	SUBFC
BCCTR	LWZX	SUBFE
BCLR	MCRF	SUBFIC
CMP	MCRXR	SUBFME
CMPI	MFCR	SUBFZE
CMPL	MTCRF	XOR
CMPLI	MTSPR	XORI
CNTLZW	MULHWU	XORIS
CRAND	MULLHW	
CRANDC	MULLI	
CREQV	MULLW	
CRNAND	NAND	
CRNOR	NEG	
CROR	NOR	
CRORC	OR	
CRXOR	ORC	
EQV	ORI	
EXTSB	ORIS	
EXTSH	RLWIMI	
ISEL	RLWINM	
LBZ	RLWNM	
LBZU	STB	

DNACore

ADD	ORI
ADDI	ROTR
ADDIU	ROTRV
ADDU	SB
AND	SEB
ANDI	SEH
B	SH
BEQ	SLL
BGEZ	SLLV
BGEZAL	SLT
BGTZ	SLTI
BLEZ	SLTIU
BLTZ	SLTU
BLTZAL	SRA
BNE	SRAV
CLO	SRL
CLZ	SRLV
EXT	SUB
INS	SUBU
J	SW
JAL	SWAT
JALR	SWATLA
JR	SWATLB
LB	SWATLH
LBU	SWATMM
LH	SWATRES
LHU	SWATSH
LUI	SWL
LW	SWR
LWL	WSBH
LWR	XOR
MOVN	XORI
MOVZ	
NOP	
NOR	
OR	

CoreVA

ABS	STB
ADC	STH
ADD	STW
AND	VMLA
ANDN	VMLS
ASR	VSMLA
BR	VPACK
BRL	VABS
CAN	VADD
CEO	VSAS
CLZ	VSSA
CMP	VASR
COR	VLSL
DEC	VLSR
DEC4	VSUB
EOR	VRSB
LDW	
LSL	
LSR	
MCR	
MLA	
MRC	
MOV	
MOVLO	
MOVHI	
MVAL	
MVAH	
MVB	
MVH	
MVSB	
MVSH	
OR	
RSB	
RSC	
SBC	
SUB	

Figure 8.2: Overview of all instructions that have been specified in ViDL.

and an extension has been added. The instruction set extension includes 7 instructions for acceleration of the Smith-Waterman algorithm. Thanks to ViDL's maintainability, the DNACore ISA has rapidly been derived from the MIPS ISA.

Besides, two academic instruction sets have been specified, namely the "one instruction set computer" (OISC) and the "simplified reduced instruction set computer" (SRC), which has been proposed by Heuring and Jordan [25].

8.2.2 Efficient specification

ViDL can be used to specify realistic instruction sets at an acceptable effort. As a result, time-to-market is improved and the costs of development are reduced. For evaluation, several existing instruction sets have been specified in ViDL. Semantics of specified instructions have been defined exactly. This is much more challenging than defining an instruction set that is tailored to the features of the specification language. The effort for specification is linear in the complexity of the instruction set, due to loose coupling in ViDL.

The specification of the *ARM ISA* comprises 143 instructions using 800 lines of code. Semantics of condition evaluation, conditional execution, auto-increment of LD/ST, and the shifter-operand have been factored out using functions. The functions are reused among instructions and can easily be changed, for instance in the scenario of design space exploration. Formalization took approximately one month and was dominated by understanding the ARM instruction set, which is described in the 800 pages comprising ISA manual.

The *MIPS ISA* was defined in one day only. The specification consists of 550 lines of ViDL code for 74 instructions. Specification of registers and memories

```
instruction jal
  encoding "000011iiiiiiiiiiiiiiiiiiiiiiii"
  semantics begin
    pc[0]<2>=cat(removeBitsRight(pc[0],28),addZerosRight(i,2));
    r[31]=add(pc[0],8);
  end
end
```

Figure 8.3: Specification of MIPS's "jump-and-link" instruction in ViDL.

(the state) accounts for another 80 lines. An instruction is typically specified using 6-7 lines, where only 1-2 lines are required to define its semantics. Figure 8.3 shows the ViDL code of a section-relative "jump-and-link" instruction, which belongs to the more complex instructions. It has one delay slot, as specified.

The ViDL specification of an *OISC* (one-instruction-set-computer) consists of 30 lines of code, including one additional instruction, which is required for initialization. The semantics of the `subleq` instruction[2] require only three lines of ViDL code. The SRC instruction set has been specified for a comparison with a respective Lisa specification (Section 3.6.6). The instruction set consists of 27 instructions and has been specified in ViDL in 90 minutes, which includes the time for reading about the instruction set. The ViDL specification consists of 207 lines of code (97 logical lines of code).

The *Power ISA* specification is by far the longest of all ViDL specifications. It includes 95 instructions and is specified by approximately 1600 lines of ViDL code. It is a strict one-to-one formalization of the Power instruction set manual. For instance, common instruction semantics have not been factored out. Instead, this task is postponed to the processor generator. The instruction set has been specified in ViDL by a computer science student, who did not have experience in ViDL and engineering of processors.

8.2.3 Usability

Compared to other specification languages, ViDL has a lower complexity. This reduces costs for training of new users and enables deployment of less experienced developers. Hence, processor and instruction set development is opened to a wider range of users. Developers greatly benefit from the expert knowledge that is encapsulated in the generators.

For evaluation, two real word instruction sets have been formalized in ViDL by students, who were previously not involved in the development of processors and instruction sets. The *Power ISA* was specified and evaluated by a computer science student in the context of his master thesis. He did not have experience in the area of instruction sets, ViDL and implementation of processors. His thesis

[2] subtract and branch on lower equal

took 6 month, including learning of ViDL and understanding the Power ISA. During this time, he has also evaluated the generated simulator, the generated processor and has written his thesis. The *CoreVA ISA* was specified by a student of electronic engineering within two months. The student did neither know ViDL, nor the CoreVA instruction set in advance.

Both students used the ViDL manual and required only little help to specify the instruction sets. The same holds for the generation of the simulator and the processor implementation. These two projects also provided new insights on how ViDL is understood and used by other developers. For instance, common computations have likely not been factored out and "expensive" or redundant operations have been applied. Fortunately, such code is optimized by the generator. Hence, inefficient specifications do likely not affect the generated products.

8.2.4 Rapid exploration of instruction sets

Using ViDL, the design space of an instruction set can rapidly be explored. Instructions can easily be removed, added or changed. This property accelerates the development of application specific instruction set processors (ASIPS) and thereby reduces development time and costs.

For evaluation, the bit-with of the entire ARM instruction set has been explored. The ARM ISA had at first been specified as described in the manual, namely as a 32-bit instruction set. It has then later been extended to a generic n-bit instruction set. Therefore, the width of the general purpose register file has been redefined. Other storages (program counter, status register and memory) remained unmodified, i.e. 32-bit wide. Instructions, which transfer data between these storages and the general purpose registers therefore had to be considered. Fortunately, these transfers have been identified and reported by the type inference as type errors. The conflicts (one for `brl` and one for `msr`) have been resolved by inserting zero-extensions. Actually, the kind of extension is a design decision of the developer and can not be automated. Since these changes, the instruction set is generic with respect to bit-widths. To change the width of the ARM specification, only the width of the general purpose registers needs to be redefined. This unplanned extension to an n-bit instruction set took about one hour. The resulting ultra-wide instruction sets are correctly implemented as simulators and processors. Evaluation results for ARM processors and ARM simulators of various widths are presented in the next sections.

8.2.5 Restrictions

ViDL is a powerful language, which can be used to precisely specify semantics of a wide range of instructions. Against the background of a PhD thesis, generators have been developed to an exceptional high level. Correct simulators and processors have been generated for a series of instruction sets. However, some kinds of instructions have not been included in the ViDL specification. In the

following, these kinds of instructions are described, along with proposed future extensions.

Coprocessor related instructions have been omitted, as they refer to an external peripheral, namely the coprocessor. To support such functions, I/O ports need to be defined in ViDL. I/O Ports (Section 4.2) can be defined in ViDL, but are not supported by the generators yet.

Memory management instructions, which control virtual memory (see [24], Chapter 5), have been omitted, as they refer to an external peripheral, namely the memory management unit (MMU). To support this feature, a (generic) MMU needs to be developed, and made accessible via I/O ports in ViDL.

VLIW/DSP instruction sets can not be defined, as ViDL does not include respective language constructs yet. An extension of ViDL may allow to define the structure of a VLIW and the relation between its slots and instructions.

Iterative semantics of instructions can not be described in ViDL yet. This affects for instance load/store multiple instructions of ARM. Support may be added in terms of a predefined "iteration" functional. The iteratively "executed" behavior could be supplied as a function to this functional.

Floating-point instructions are omitted, as respective primitives have not been defined yet. To support floating point instructions, a respective set of primitives needs to be added to the primitive library.

Integer division instructions have been omitted, as respective primitives have not been defined yet. Division instructions can be supported by adding division or division-step primitives to the primitive library.

Delay slots of branches are not implemented in processors, as the generator does not support this feature yet. However, the simulator generator supports delayed branches. To add support to the processor generator, the construction algorithm for branch prediction needs to be extended.

Variable length instructions are not supported by the generator yet. To support such instructions, the construction of instruction fetch logic needs to be extended.

Hybrid 16/32 bit wide instruction sets (e.g. ARM Thumb or MIPS16) are used in embedded systems, to reduce code size. Such 16-bit extensions can conveniently be specified in ViDL, although they overlap with 32-bit instructions and require reconfiguration of the instruction decoder. However, they involve variable length instructions and are therefore not supported by generators yet.

Embedding of the program counter into ARM's register file has been omitted, as only one storage per architectural interface is supported by the generator yet.

The solution for most of these restrictions is considered a matter of programming. Although this list is quite long, these restrictions only have an impact on exceptional and very few "general purpose" instructions. Almost all data processing instructions, control instructions and load/store instructions have exactly been specified, without any modification of semantics.

8.3 Generator speed

The simulator generator and the processor generator are fast. The speed of a generator is a major objective in practice. A generator that takes one day to produce a simulator or processor will hardly be accepted by users, as it defers development and makes iterative improvements virtually impossible. During development of generators, care has been taken to implement efficient data structures and algorithms. Using profiling tools, like Valgrind, bottle necks have been detected and eliminated in both generators. As a result, generating a processor or simulator is a matter of minutes or even seconds.

To evaluate the speed of the generators, many simulators and processors have been generated. Generating simulators of 13 different instruction sets has taken 3.5 minutes, i.e. 17 seconds per simulator on average. The longest generation took 26 seconds for a 257-bit wide ARM simulator and the fastest (OISC) was finished after less than 1 second. The generation of the 83 processors that have been used for all evaluations took 118 minutes, i.e. 85 seconds per processor on average. The generation time significantly depends on the complexity of the instruction set. Generating a processor for OISC takes 1 second, whereas the longest time of 4 minutes is required for an ARM processor. As generation is fast, modified instruction sets can rapidly be evaluated. This allows for iterative development processes and exploration of the instruction set's design space.

8.4 Simulator generator

A fast instruction set simulator is crucial for profiling of real word applications and for testing the specification by simulating an extensive set of test programs. The main objective of this section is therefore to show, that generated simulators are fast and that the implemented generation methods are effective. For evaluation, the ARM instruction set is regarded. The generated simulators of the other instruction sets yield similar results.

After describing the evaluation environment (Section 8.4.1), characteristic classes of ARM instructions are introduced, which significantly differ in simulation speed (Section 8.4.2). Benchmarks for these classes are then used to evaluate simulation speed. It is shown, that simulation is also fast for ultra-wide instruction sets (Section 8.4.3), by simulating 64-bit to 256-bit wide variants

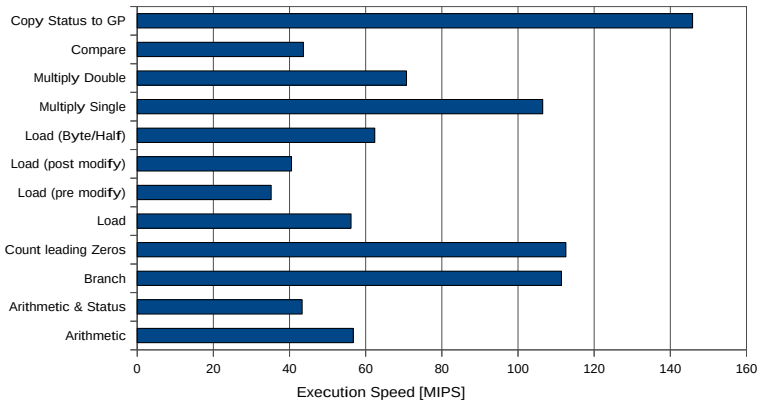


Figure 8.4: Simulation speed in million instructions per second (Mips) for characteristic instruction classes.

of the ARM ISA. Tailoring the simulator implementation to the width of the host computer improves simulation speed (Section 8.4.4), as shown for 32-bit and 64-bit wide C-code. Optimizations that are implemented in the simulator generator significantly accelerate simulation (Section 8.4.5). For evaluation, each optimization has been deactivated separately. The speed of the resulting simulator has then been compared to the fully optimized simulator.

8.4.1 Setup

The ISS has been evaluated on an Intel® Core™2 Duo E8400 machine at 3 GHz running Linux. The generated ISS code has been compiled using `GCC-4.4.5` with optimization level `-O3` turned on. To measure simulation time, the `time` command has been used.

To measure the simulation speed of a specific class of instructions, dedicated benchmarks have been created, in which the respective instructions are executed repeatedly. For all simulations, at least 10^8 ISA instructions have been executed. Execution time for simulator initialization and control instructions in the benchmark are insignificant.

8.4.2 Characteristic instructions

The semantics of an instruction have a significant impact on its simulation speed. In the following, classes of similar instructions are evaluated with respect to simulation speed. To evaluate the simulation speed, an optimized 32-bit simulator is generated for the original ARM ISA. Neither the generated simulator code, nor the ARM ISA have manually been tuned in any way. Figure 8.4 shows the simulation speed in million ARM instructions per second (Mips) for characteristic instruction classes. The simulation speed highly depends on the executed

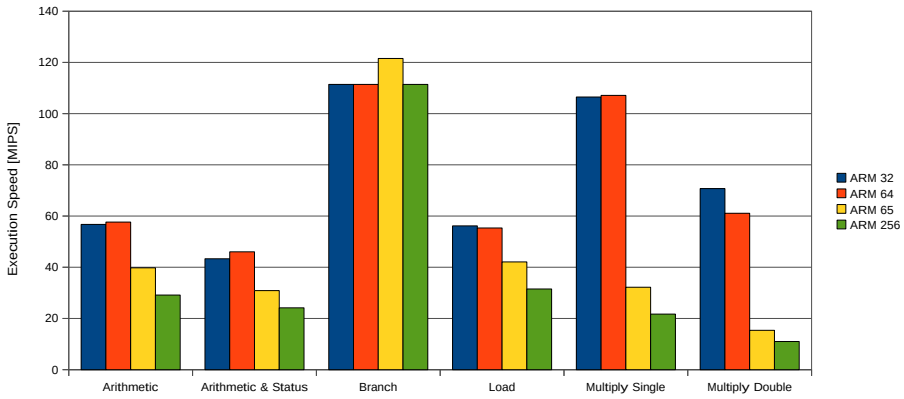


Figure 8.5: Effect of ISA width on simulation speed.

instruction, ranging from 35 Mips to 140 Mips. The simulation of branch, multiply, count-leading-zeros and copy-status-register instructions is rather fast. These instructions do not use the shifter-operand of the ARM architecture, which consumes much simulation time.

The simulation speed of load instructions lies between 55 Mips for plain loads, 35 Mips for pre-incremented loads and 40 Mips for post-incremented loads. The incrementing load instructions also modify the base register, which slightly reduces simulation speed. There is also a notable difference between arithmetic instructions and arithmetic instructions, which set the status flags in addition. The evaluation of the status flags (carry, negative, zero and overflow) is expensive and reduces simulation speed by about 25%. In summary, there may be a large gap between the simulation speed of an instruction and its complexity of its hardware implementation. Simple operations that are executed concurrently in hardware need to be simulated sequentially, which has a major impact on speed. A multiplication on the other hand is complex in terms of hardware, but can efficiently be simulated.

8.4.3 ISA width

Wide instructions are efficiently simulated, i.e. the simulation speed decreases only slightly with the width of operations. For evaluation, an ARM instruction set has been instantiated for a datapath width of 32, 64, 65, and 256 bits. The ISS has been generated and compiled for a 64-bit system, i.e. slots are 64 bit wide. Figure 8.5 shows, that the simulation speed for the 32 and 64 bit ARM ISAs is approximately the same. A bit-string of the ARM fits into a single 64-bit slot and therefore only single integer operations are required for evaluation. The only exception is the double multiplication, which opposed to other instruction yields a 128 bit wide result for the 64-bit ARM ISA. To compute the 128 bit result, 4 multiplications must be performed on the host,

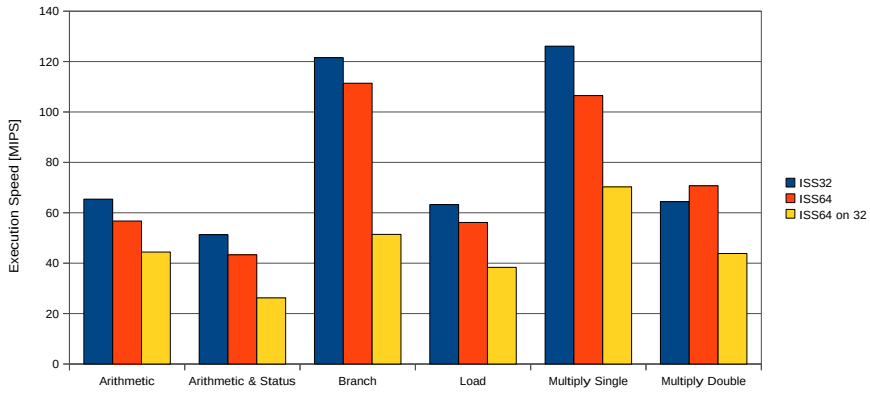


Figure 8.6: Effect of simulator-width on simulation speed.

which decreases simulation speed.

By increasing the ISA width beyond 64 bits, bit-strings need to be stored in more than one slot. Thereby a series of 64-bit integer operations is required to evaluate one n-bit ($n > 64$) primitive. For the 65-bit ARM, two 64-bit slots are used to hold a bit-string. The ISS thereby operates on 128 bit data, by applying a series of integer operations. As the 63 most significant bits of the high order slot are undefined, the ISS masks intermediate results, where necessary.

The simulation speed of the branch instruction is not affected, as the width of the program counter has not been modified. The branch target is computed using 32 bit arithmetic.

As one may expect, widening the ARM ISA has a major effect on multiplication instructions, as the number of 64-bit multiplications grows quadratically in the number of slots. Although wide multiplication is complex, the generated ISS simulates 10^7 256/512-bit multiplication instructions per second.

Evaluation shows, that the generated simulator is very fast, even for exceptionally wide instruction sets. Instructions that are not affected by widening (e.g. branch) retain a high simulation speed. Only those instructions that have to perform wide computations are affected. Wide arithmetic is efficiently broken down to host integer arithmetic. Hence, ViDL offers the comfort of arbitrary precision libraries at the speed of plain integer C-code.

8.4.4 Width of simulator code

The simulator generator can produce C-code of different widths (e.g. 32-bit and 64-bit). This width can be configured by the developer at generation time. Tailoring the width to the host system yields significant speed improvements. For the 32-bit ARM ISA, best results are yielded for 32-bit code, as shown in Figure 8.6. Generating 64-bit ISS code results in additional masking and therefore even slows down simulation. The long multiplication benefits from

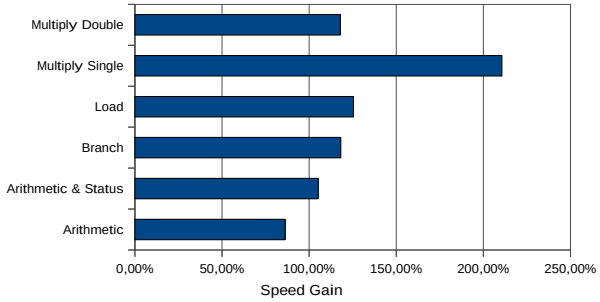


Figure 8.7: Effect of lazy evaluation on simulation speed.

64-bit wide ISS code, as it computes a 64-bit wide result. As the figure shows, compiling 64-bit ISS code for a 32-bit machine leads to a slow simulation, as 64-bit integer arithmetic in the ISS code can not efficiently be executed on a 32-bit host.

8.4.5 Generator optimizations

The ISS generator performs a number of transformations and optimizations to produce efficient ISS code. The effect of some of these optimizations on the simulation speed is evaluated in the following. For the evaluation of an optimization X, the fully optimized ISS is compared to the fully optimized ISS without optimization X.

8.4.5.1 Evaluation strategy

The ISS generator can generate ISS code with an eager or lazy evaluation strategy, where lazy evaluation significantly increases simulation speed. Using eager evaluation, all operations are evaluated in a single bottom-up pass. Each operation is evaluated exactly once and the result is then reused. The latter is valid, as primitives are free of side-effects. Using eager evaluation, a primitive is always evaluated, whether the result is used or not. In contrast, lazy evaluation computes the result of a primitive only, if it is actually used, i.e. when it affects the processor state. The implementation of lazy evaluation uses a recursive bottom up evaluation with caching of previously computed results. Figure 8.7 shows the speed gain by using lazy evaluation, instead of eager evaluation. In average, the speed is increased by 120%. As the multiplication instruction is already simulated efficiently, the relative speed gain becomes even clearer (210%).

8.4.5.2 Instruction decode cache

Using a decode cache, an instruction is decoded at most once. The association between instruction address and instruction is then cached and reused throughout simulation. The decode cache significantly increases simulation speed, es-

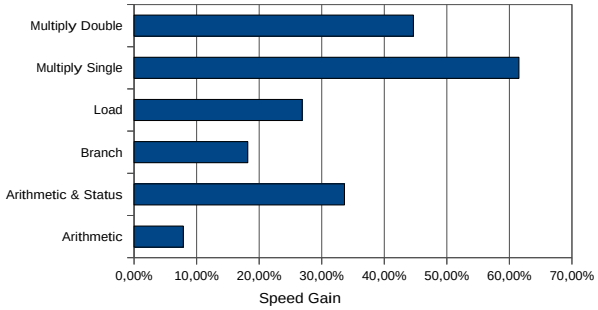


Figure 8.8: Effect of decode cache on simulation speed.

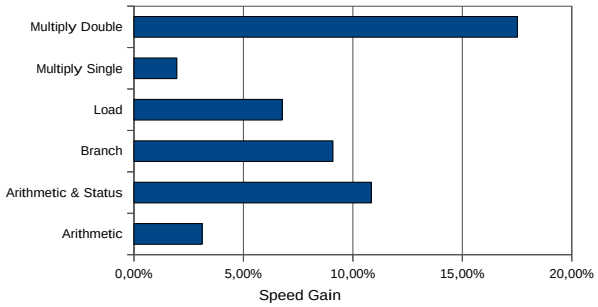


Figure 8.9: Effect of rewriting on simulation speed.

pecially of instructions that are dominated by decoding. Figure 8.8 shows the speed gain by using a decode cache. Depending on the class of instructions, a speedup of up to 60% is reached. Especially instructions that are dominated by decoding benefit from a decode cache. The multiplication instructions for instance are efficiently simulated, but have a depth of 7 in the decision tree, which means that the ISS has to perform approximately 7 branches[3] to decode the instruction.

8.4.5.3 Term rewriting

Term rewriting is used in the generator to simplify the DFG, for example by strength reduction or partial evaluation. Term rewriting leads to more efficient C-code and a faster simulator. For the instruction classes in Figure 8.9, a speed gain between 4% and 17% is reached. For some instructions, term rewriting does not seem to have an effect. However, this does not mean that the ISS generator did not enhance the ISS-code. It shows that some simplifications done by the ISS generator can equally be performed by the C-compiler. As

[3]This is the nesting depth of "if-then-else" and "switch" statements in the generated C-Code. The actual number of branches also depends on compiler optimizations.

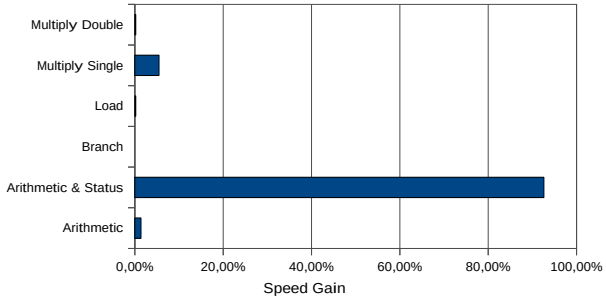


Figure 8.10: Effect of write merging on simulation speed.

mentioned, the ISS-code is compiled with a high optimization level (`-O3`). If for instance constant expressions are not evaluated by the ISS generator, they are evaluated by the C-compiler and therefore the optimization of the generator does not become visible. A speed gain therefore means that the ISS generator applied optimizations that can not be performed by the C-compiler.

8.4.5.4 Merging of write accesses

The simulator generator includes a "Write Merge" optimization, which detects and merges write accesses to the same storage. For the ARM ISA for instance, multiple write accesses to separate status bits are merged to a single write access to the whole status register. The diagram in Figure 8.10 clearly shows that instructions that set status bits are accelerated by this optimization. The execution speed of these instructions is increased by 92%. Instead of performing 4 sequential write accesses to the status word for each status flag (C,N,Z,V), only one merged write access is performed. This reduces the overhead for masking and the number of transactions by a factor of 4. The remaining instructions are not affected by the optimization and the speed is therefore nearly unchanged.

8.4.5.5 Summary

It has been shown, that the optimizations of the generator are essential for efficient simulation. Omitting one or more of these optimizations decreases simulation speed notably. The ISS reaches a simulation speed of 60 Mips to 120 Mips on an Intel Pentium 3 GHz desktop processor. The simulator is implemented using platform independent standard C-code, i.e. the code does not include inline assembly or compiler specific extensions to further increase the simulation speed. Neither the ViDL specification of ARM nor the generated simulator code has manually been tuned to increase simulation speed.

Compared to other interpretative simulators ($\approx$ 10 Mips), the generated simulator is quite fast ($\approx$ 60 Mips). However, it is slower than simulators that apply just in time compilation and binary translation. The latter reach simulation

speeds of 200 Mips to 800 Mips. Generating such a simulator is considered future work.

8.5 Processor generator

This section evaluates the quality of generated processors with respect to physical and dynamic characteristics. The former includes the estimated clock frequency, chip area, and power consumption of a physical processor. These figures have a major impact on production costs and the application area of the processor. For instance, a processor with high power consumption is not suited for mobile applications. Dynamic characteristics of a processor comprise pipeline stalls and penalties due to misprediction. These figures determine how efficient a certain program can be executed. For instance, long instruction latencies and high penalties likely lead to a much lower number of executed instructions compared to the processors clock frequency.

To give a basic impression of the generator's quality, processors of major instruction sets are presented first (Section 8.5.2). The microarchitectural design space of these processor is explored afterwards (Section 8.5.3), to demonstrate the benefits of ViDL and the flexibility of the processor generator. It is also shown that the *target frequency* parameter (Section 7.6.3) effectively guides decisions of the processor generator on microarchitectural aspects. A comparison between generated and handcrafted processors is presented using the example of the CoreVA instruction set (Section 8.5.4). The scalability of the processor generator with respect to exceptional instruction sets is examined at the example of OISC (Section 8.5.5) and ultra-wide ARM instruction sets (Section 8.5.6). After evaluating physical characteristics of generated processors, the microarchitecture is discussed. In particular, the number of ports (Section 8.5.7) and the pipeline structure is regarded (Section 8.5.8), to show that the generated microarchitecture is reasonable. To examine dynamic characteristics, the effect of different microarchitectures on instruction latencies and branch penalties is evaluated (Section 8.5.9). Finally, the resolution of data-hazards and control-hazards is demonstrated, by executing programs on the generated processor in ModelSim. The resulting waveforms are analyzed with respect to correctness and efficiency (Section 8.5.10). The setup for evaluation is briefly described in the next section.

8.5.1 Setup

Processors have been generated on a 2.6 GHz *server system*[4]. The generated VHDL code uses simple VHDL primitives and *Synopsys DesignWare datapath and building block IP*. The generated processors have been tested using *ModelSim 6.4*. Therefore, test programs haven been loaded into the memory of the simulated processor and executed. Waveforms of registers and pipeline-control

[4] 32-bit Linux; 2 Intel Xeon E5430 processors; 16GiB of RAM

signals have been checked. A waveform shows the state of a digital signal over time, similar to an oscilloscope.

Processors have been synthesized using *Encounter RTL Compiler v8.10* from Cadence. Synthesis was performed for *STMicroelectronics* 65 nm *low power standard cell* technology. A synthesis breaks the register-transfer level description (VHDL) down to a technology specific netlist of standard cells. A standard cell is a low level implementation of a digital building block, such as a logical NAND-gate, a 2-bit full adder or a flip-flop. The netlist defines the set of standard cells and their interconnection. Synthesis has been performed for worst case conditions, which means that the digital circuit has been dimensioned for a lowered supply voltages of $1.1\,V$ and a chip temperature of 125°C. Worst case conditions have a negative impact on synthesis results, but ensure correct operation of the final chip, which may then even be run at a higher clock frequency.

Synthesis has been performed for the entire processor core, i.e. all functional units and all registers. Data- and instruction-memory have been omitted, to evaluate the generated core only. The process for synthesis is described in Section 8.1. Each processor has been synthesized with a very tight timing constraint of 1 ns (corresponds to 1 GHz) first, to determine the critical path and the related feasible timing $t_{\min} = 1\,\text{ns} - \text{SLACK}$. To maximize clock frequency in a second pass, synthesis has been constrained for $1.05 * t_{\min}$, i.e. almost the lowest feasible timing. Besides figures on clock frequency, synthesis also estimates power consumption and chip area. Estimation of power consumption is based on a statistical switching probability of 0.2, which has proven to be a good approximation in practice. Area estimations include the precise area of standard cells and estimated area for routing.

8.5.2 Overview of generated processors

In the following, physical characteristics (clock, power, area) of generated processors are presented. A comparison to a handcrafted processor is given later in Section 8.5.4.

Figure 8.11 shows the synthesis results for ARM, MIPS, Power, CoreVA, DNACore and OISC. The ARM and MIPS processors have been generated for a high target frequency of 500 MHz, Power for 400 MHz and DNACore and OISC for 100 MHz. The target frequency and its impact on clock, power and area is discussed in the next section. Under worst case conditions, all generated processors reach a *clock frequency* of more than 600 MHz. The frequency can be considered as high, especially since a low power technology has been used. The clock frequency of the OISC exceeds 1 GHz. It is limited by a 32-bit adder for computation of the next instruction address. The estimated *power consumption* lies in the range of 20 mW to 160 mW. Surprisingly, the generated ARM processor has the highest power consumption, although it is known as a processor for low power applications. The high power consumption is a result from deep pipelining, which is itself a result of the high target frequency (500 MHz). Reducing the target frequency leads to a shorter pipeline and significantly reduces power consumption. As shown later the power consumption of the ARM pro-

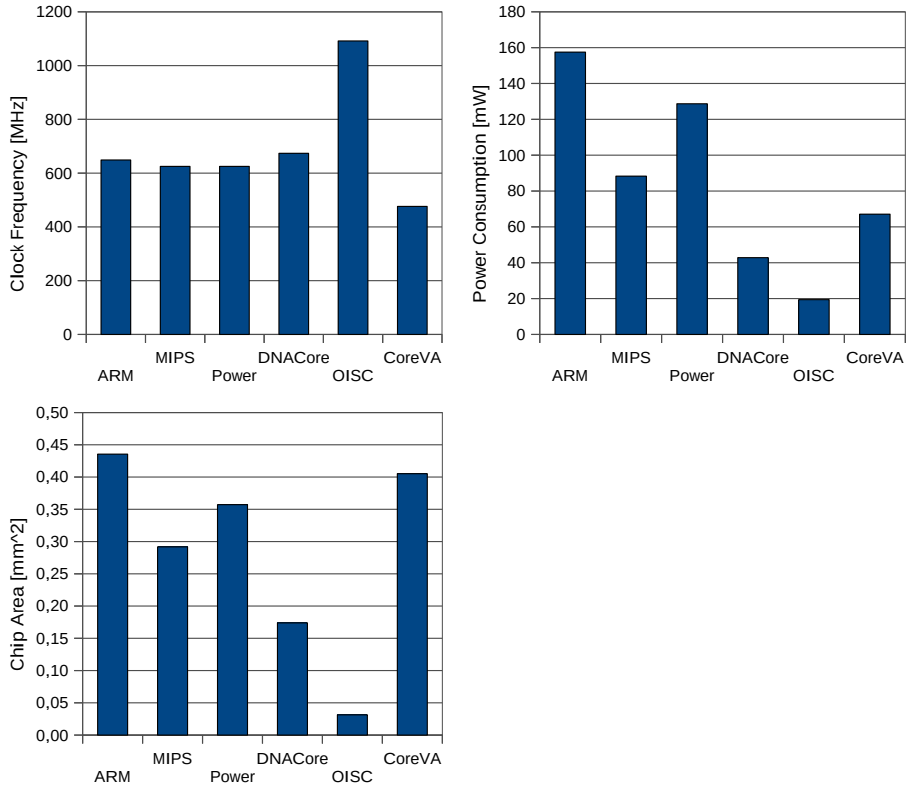


Figure 8.11: Clock frequency, power consumption and chip area of generated processors.

cessor can be reduced to 28 mW for a target frequency of 200 MHz. It should further be reduced by enabling clock gating, which likely reduces power consumption by 50%. The effects of target frequency are discussed in the next Section. As Figure 8.11 shows, the estimated *area requirements* lie in a range of $0.03 - 0.45\text{mm}^2$. It is nearly proportional to power consumption. As for power consumption, chip area can significantly be reduced, by generating for lower target frequencies.

Unfortunately, the results for ARM, Power and MIPS can not be compared to handcrafted VHDL code, as such implementations are not available. Respective synthesis results of the processor cores for different technologies appear to be a well kept secret. For instance, the ARM company only publishes *relative figures*, such as "Variant X is 20% faster then variant Y". Fortunately, a handcrafted VHDL implementation of the CoreVA processor has been available for a comparison (Section 8.5.4).

8.5.3 Exploration of microarchitecture

In the following, the impact of the target frequency on clock frequency, power consumption and chip area of the generated processors is evaluated. Therefore, at least 5 processors have been generated for each instruction set. The target frequency for generation ranges from 100 MHz to 700 MHz, in steps of 100 MHz. For each target frequency, a processor implementation with a dedicated microarchitecture has been generated. In the following, a processor is denoted by the name of its instruction set and the target frequency. For instance, the `ARM_F300` processor has been generated for a target frequency of 300 MHz. All processors of one instruction set have been generated from the very same ViDL specification. Only the target frequency parameter has been varied. This allows for automatic exploration of the microarchitectural design space.

As Figure 8.12 shows, the target frequency that is passed to the processor generator successfully guides generation. Remind that the generator aims to construct the microarchitecture such, that the target frequency is met, while minimizing latencies and hardware resources.

With an increasing target frequency, the generator reduces the critical path by deepening the pipeline and by deferring forwarding. This in turn leads to a higher feasible *clock frequency*, of the synthesized processor. The highest gain for ARM, MIPS and Power is encountered between 200 MHz and 500 MHz. The gain is mainly caused by pipelining of the multiplier. For 200 MHz, the generator instantiates a non-pipelined multiplier, for 300 MHz a 2 stage multiplier and for 500 MHz a three stage multiplier. In contrast to generated ARM, MIPS and Power processors, the clock frequency of DNACore processors is nearly constant, as its ISA does not include multiplication instructions.

As you can see in the Figure, the maximal clock frequency of the synthesized processor lies approximately 100 MHz over the user supplied target frequency. The generator estimates the critical path quite conservatively. The generator may be adjusted, such that the relation between both frequencies is nearly linear.

Increasing the target frequency leads to longer pipelines. As a result, chip

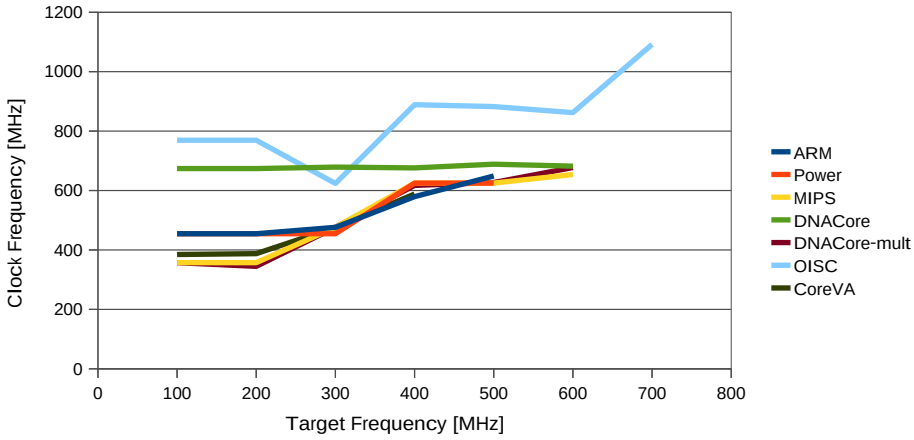


Figure 8.12: Relation between targeted frequency and achieved clock frequency of synthesized processors.

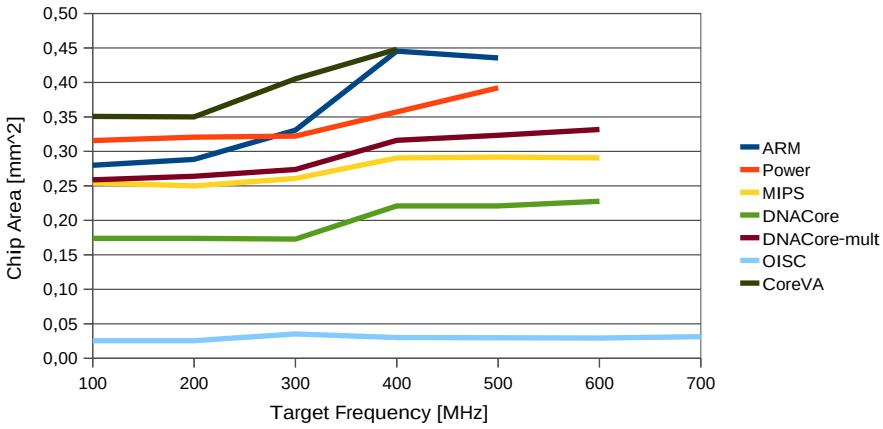


Figure 8.13: Effect of target frequency on chip area.

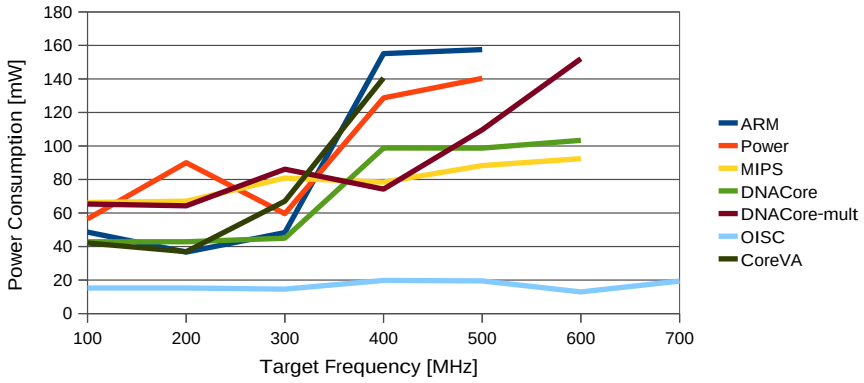


Figure 8.14: Effect of target frequency on power consumption.

area and power consumption increase, as shown in Figure 8.13 and Figure 8.14. A major rise in power consumption can be seen between 300 MHz and 400 MHz. This can be attributed to additional resources for pipelining, as well as tighter timing constraints for synthesis. Remind that a timing constraint is supplied to the synthesis tool (Figure 8.1), which is not related to the target frequency of the processor generator. For most ISAs, the synthesis timing constraint is reduced by approximately 1 ns from 2.8 ns (360 MHz) to 1.8 ns (625 MHz). Lowering the timing constraint (synthesis) of `ARM_F500` from 1.7 ns to 1.4 ns increases power consumption by 23%, although the RTL description remains unchanged. With respect to the microarchitecture, the generated forwarding circuit of the `ARM_F500` processor accounts for 25% of power consumption.

It has been shown that the microarchitectural design space can automatically be explored using ViDL and its processor generator. The instruction set is specified once and a series of different processors with very different physical characteristics is generated at no additional effort. According to the area of application and timing requirements, the developer can select one of the generated processors. For instance, the generated `ARM_F300` processor is energy efficient, whereas the `ARM_F500` processor maximizes performance. The target frequency does not only affect physical characteristics of the processor, but also its dynamic behavior, which is evaluated in Section 8.5.9.

8.5.4 Comparison to handcrafted processors

So far, synthesis results of various generated processors have been discussed. To estimate the quality of the processor generator, these results need to be compared to results for handcrafted and optimized VHDL code. However, results for ARM, MIPS and Power can not be compared. Neither are respective HDL implementations freely available, nor have synthesis results been published for the processor core and similar chip technologies. Most publications on these commercial processors only make *relative* statements on physical characteris-

tics. Fortunately, handcrafted and optimized VHDL code has been available for the CoreVA[5] processor. This section therefore compares this implementation to the generated CoreVA processors.

In the following, minor differences in the handcrafted and the generated processors are explained. Physical and dynamic characteristics of processors are then compared. Finally, the specification time for ViDL is related to that for manual development of VHDL code.

The comparison between handcrafted and generated processors for CoreVA is fair. The same synthesis tools have been used with the same configuration and the same chip technology[6] has been targeted. Synthesis results consider the same part of the processor, i.e. the core, including all registers, but excluding instruction- and data-memory. The CoreVA instruction set has completely been specified, except for three aspects.

- Branches have no delay slots.
- Division instructions are not specified.
- The 64-bit wide move-immediate instruction has been replaced by a "move high" instruction and a "move low" instruction.

These aspects should have a minor impact on synthesis results. The hardware implementation of the 64-bit move instruction is very simple. The SRT division step-unit accounts for less than one 1% of chip area. It does not contribute to the critical path and does therefore not affect the processor's frequency. The non-delayed branch even introduces a tighter timing for the computation of the target address and the condition. One can therefore assume that the modifications barely improve synthesis results of the generated processors.

Figure 8.15 shows the synthesis results for the handcrafted CoreVA processor and 5 generated implementations. For all processors, clock gating has been activated for synthesis. Clock gating is a method to reduce power consumption of digital circuits, by deactivating unused parts, at the expense of an additional propagation delay. It can automatically be added by synthesis tools.

As Figure 8.15 shows, the pipeline depth of the generated processors ranges from 4 stages to 7 stages, increasing with target frequency. For a target frequency of 300 MHz, the generated pipeline matches the depth of the handcrafted one. The clock frequency, power consumption and area requirements of generated processors increase with pipeline depth. The clock frequencies of generated processors lie in the order of the handcrafted implementation. The 4-stage processor `CoreVA_F100/nRx` (target frequency 100 MHz; no bypass-relaxing) is 18% slower and the 7-stage processor `CoreVA_F400` is 37% faster compared to the handcrafted implementation. Considering the same pipeline depth, the `CoreVA_F300` processor faster, but requires approximately 3 times as much power and area. The high values for power and area may be attributed to the simple hardware sharing method that is implemented in the generator.

[5] scalar version (not VLIW)
[6] STMicroelectronics 65 nm low power

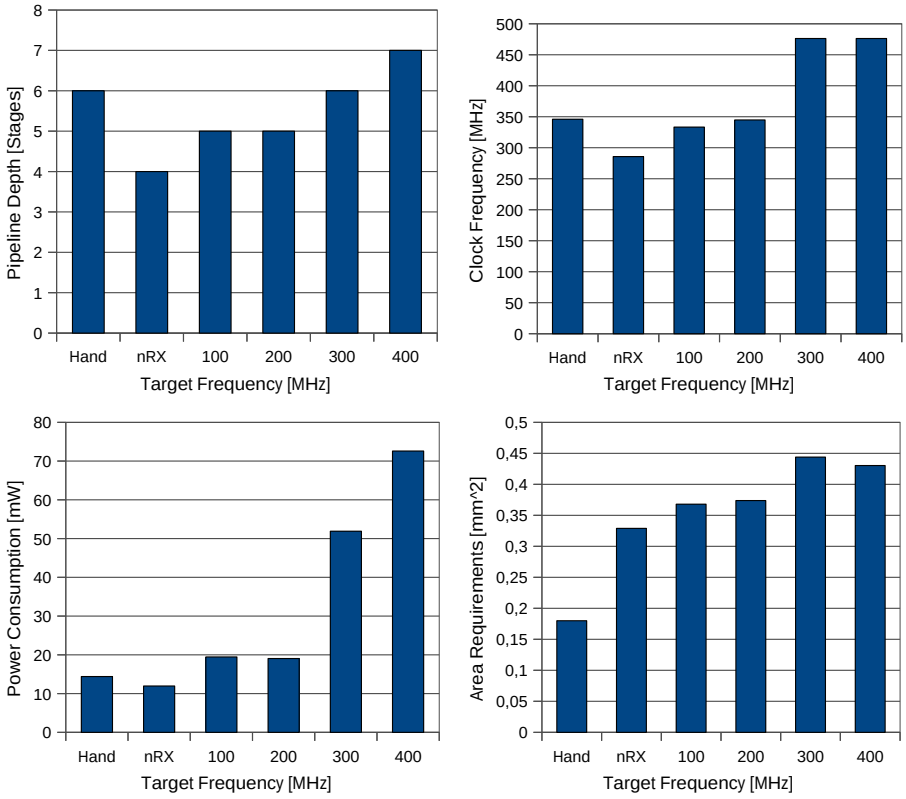


Figure 8.15: Clock frequency, power consumption and chip area of a handcrafted and multiple generated CoreVA processors for activated clock gating.

More sophisticated methods can significantly increase the degree of sharing and thereby reduce area as well as power consumption.

The dynamic behavior of the generated CoreVA processors is similar to the handcrafted one. The latter has a latency of 0 for nearly all (95%) instructions, except for multiplication- and load-instructions. For the `CoreVA_F100/nRx` processor, all instructions have a latency of zero and branches have no penalty. With increasing pipeline depth, the number of zero-latency instructions decreases to 73% for the 7-stage `CoreVA_F400` processor.

Specifying the instruction set by a student took two month, compared to approximately one year for a manual processor implementation by an experienced VHDL developer. Thanks to generation, testing of generated pipeline control can be omitted. Besides, a compatible simulator has been generated at no extra effort. Correctness of the ViDL specification can easily and efficiently be tested using this simulator. The microarchitectural design space has automatically been explored, by generating processors with pipelines from 4 to 7 stages.

Summing up, processors of similar performance (by means of clock frequency and CPI) have been generated in a very short time. Power consumption and area requirement are comparably high, but are expected to be reduced in future by hardware sharing methods.

8.5.5 OISC — A simple processor

The complexity of generated processors scales well with the complexity of instruction sets. In particular, simple instruction sets lead to small and fast implementations. The entire datapath, including ALU, register ports and forwarding is tailored to the semantics of instructions. To examine the relation between instruction set and generated processor, an OISC (Section 2.1.3) is evaluated in the following.

As presented in the previous section, the power consumption and chip area of OISC processors is low, while reaching high clock frequencies. For `OISC_F100`, the register file accounts for 78% of chip area and 73% of power consumption. The remainder is mostly consumed by a 32-bit adder. The `OISC_F100` processor does not include much overhead, such as unused functionality of an ALU or excessive register ports.

8.5.6 Wide instruction sets

This section briefly evaluates processors for ARM instruction sets of different widths. ARM instruction sets of different widths have already been explored in the context of the simulator (8.4.3). Figure 8.16 shows the synthesis results for `ARM_F500` processors with a width from 32-bits to 257-bits. The width of 257 is not a typo, but has been chosen as it is a prime number. Chip area rises and the clock frequency falls nearly linear in the number bits. The results also demonstrate that the generator can cope with very large and exceptional widths. In addition, it shows, that all bit-widths can statically be determined

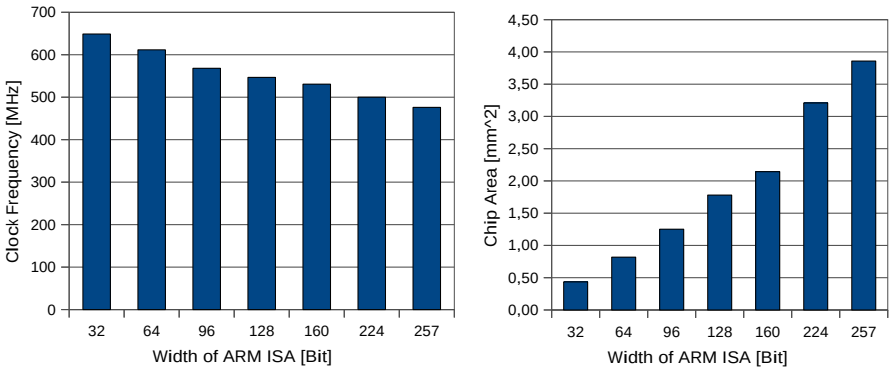


Figure 8.16: Effect of width of ARM instruction sets on processor frequency and chip area.

by type inference, as required to generate VHDL code. This feature can not be taken for granted, as the discussion of Lisa (3.6) has shown.

8.5.7 Register ports

Register ports are not specified explicitly by the developer, but derived by the generator. Since register ports are expensive in terms of chip area and power consumption, the generator aims to minimize their number. For instance, the generator typically binds mutually exclusive accesses to the same port. Even non-exclusive accesses are bound to the same port, if the generator can prove that they always refer to the same address. As a result, single registers always have only one read port and one write port.

For all evaluated instruction sets, the number of ports has been minimized by the generator, as shown in Figure 8.17. For each instruction set and storage, a record of the form `S:Rx/Wy` is given, where `S` is the name of a storage, `x` is the number of read ports and `y` is the number of write ports. For instance, the register file `r` of ARM has 4 read ports and 2 write ports. Single registers have only one read and one write port, as all accesses have been merged. The instruction memory IMEM does not have a write port, as it is only read. The number of ports of the general purpose register file is most interesting. For the ARM instruction set, the generator instantiates a register file with 4 read and 2 write ports. These ports are required to avoid conflicts for 64-bit multiplications, which read two register pairs and write one register-pair at a time. After removing multiplication instructions, the number of read ports drops to 3. Again, this is the minimal number of read ports, as arithmetic instructions require one port for the first operand and two ports for the shifter operand. For some addressing modes (e.g. `<Rm>, LSL <Rs>`), the shifter operand accesses two registers. Two write ports are required, as load instructions write the loaded value and may modify the base register in some addressing modes. Generated DNACore and

```
--- ARM
IMEM:R1/W0 r:R4/W2    cpsr:R1/W1 spsr:R1/W1 pc:R1/W1   mem:R1/W1

--- ARM-nomult
IMEM:R1/W0 r:R3/W2    cpsr:R1/W1 spsr:R1/W1 pc:R1/W1   mem:R1/W1

--- DNA Core
IMEM:R1/W0 gpr:R2/W1  swa:R1/W1  swb:R1/W1  swh1:R1/W1 swh2:R1/W1
swh3:R1/W1 swh4:R1/W1 swm:R1/W1  pc:R1/W1   mem:R1/W1

--- MIPS
IMEM:R1/W0 gpr:R2/W1  hi:R1/W1   lo:R1/W1   pc:R1/W1   mem:R1/W1

--- Power
IMEM:R1/W0 mem:R1/W1  pc:R1/W1   CR:R1/W1   LR:R1/W1   CTR:R1/W1
GPR:R3/W2  XER:R1/W1
```

Figure 8.17: Number of generated register ports for major instruction sets.

MIPS processors include a register file with two read and one write port, as it is expected for 3-address instructions. The 64-bit multiplication instructions do not introduce further ports, as they have 32-bit source operands and store their 64-bit result in the dedicated `hi`/`lo` registers.

Provided that all resource hazards are eliminated by the generator, the number of register ports is minimal. This also holds true for realistic instruction sets, which involve exceptional addressing modes and may include multiple references to the same register. The automated adaption of ports has been demonstrated by removing ARM's multiplication instructions. No other part of the ViDL specification had to be changed, as ViDL completely abstracts from register ports.

8.5.8 Structure of generated pipeline

This section presents an in-depth evaluation of generated pipelines. It focuses on the MIPS instruction set, as evaluation is very comprehensive and time consuming. The MIPS instruction set has some nice peculiarities, such as a dedicated register for multiply-accumulate instructions. Figure 8.18 is meant to give an impression of the complete pipelined dataflow graph for MIPS. The graph has been exported by the processor generator and layouted using the `dot` tool from the graphviz graph layout package. As it is quite large, a more abstract visualization is used in the following, which focuses on storage accesses and significant data-dataflow.

Figure 8.19 shows the structures of two generated pipelines. Their configuration and the generated schedule are shown below the diagrams. The *configuration* is supplied by the user and basically guides the generator. It significantly

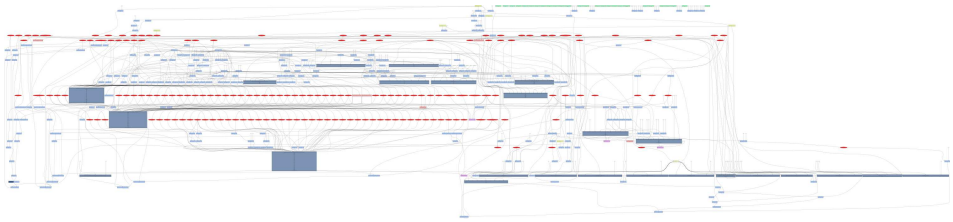


Figure 8.18: Dataflow graph of pipelined MIPS processor, as used for VHDL code generation.

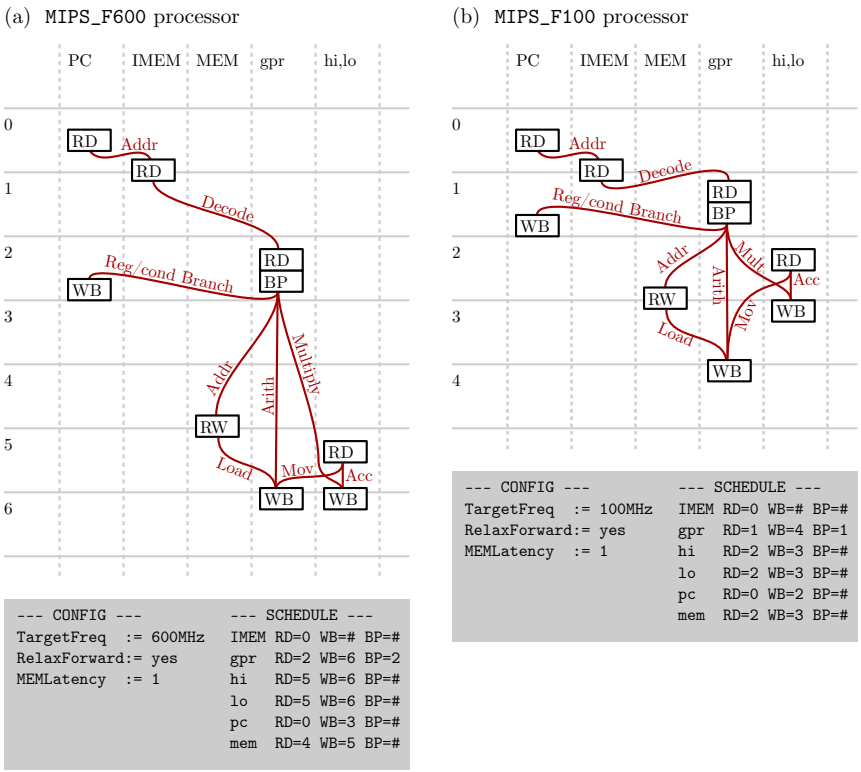


Figure 8.19: Structure of generated MIPS pipeline for a target frequency of (a) 600 MHz and (b) 100 MHz.

affects the structure of the generated pipeline. The adjoining *schedule* information shows the read stage, the write stage and the optional bypass stage for each storage. For instance, for a target frequency of 600 MHz, the general purpose registers (`gpr`) are read in stage 2 and written back in stage 6. The bypasses of the forwarding circuit are joined in stage 2. For the other storages, forwarding is omitted, as the distance between read and write stage is 1. Note, that the program counter is regarded as register for the task of scheduling.

The *illustration* of the pipeline structure in Figure 8.19 shows the location of ports and intervening dataflow. Each row in the figure corresponds to one pipeline stage, which are numbered from 0 through 6. Each column is associated with one storage. The storages `hi` and `lo` use a common column, to simplify visualization.

In the following, it is shown, that the generated pipeline has a short length and avoids forwarding in consideration of dataflow dependencies (Section 8.5.8.1), propagation delays and latencies (Section 8.5.8.2), as well as the user supplied target frequency (Section 8.5.8.3). To demonstrate scalability of the processor generator, a very short pipeline (Section 8.5.8.4) and a non-pipelined processor (Section 8.5.8.5) are presented. Finally, a generated MIPS pipeline is compared to a pipeline that is proposed by Hennessy and Patterson (Section 8.5.8.6).

8.5.8.1 Dataflow and its dependencies

In the following, the dataflow (red lines) in Figure 8.19 is briefly explained. For the sake of simplicity, irrelevant dataflow has been omitted. In the first stage (stage 0), the program counter is read. The resulting instruction address is used to index the instruction memory (IMEM) in stage 0/1. The instruction word from IMEM is decoded, yielding a set of operands. The operands are (for instance) used to index the general purpose register file in stage 2. The values from the register file are used by register branches and conditional branches, which set the program counter in stage 3. Besides, values from `gpr` are (indirectly) used to address the data memory (MEM) and to execute arithmetic instructions as well as multiplications. Multiplications write their results to `hi/lo` registers in stage 6. Multiply-accumulate instructions also read these registers for accumulation. The "move from high/low" instructions (`mfhi/mflo`) transfer values to the general purpose register file, which is written in stage 6. Besides, the `gpr` register file is written by arithmetic and load instructions.

The scheduler of the generator obeys this dataflow to determine the read stage and the write stage of each storage (Section 7.6.4). For instance, due to `mfhi/mflo` instructions the general purpose registers must be written after the `hi/lo` registers have been read. On the other hand, the `hi/lo` register can be read after the memory is accessed, since there are no "store high/low" instructions. Hence, the scheduler is sensitive to dataflow between storages.

8.5.8.2 Propagation delay and latencies

The scheduler yields a short pipeline, while considering the propagation delays and latencies of functional units. For instance, the pipelined multiplication unit in Figure 8.19a has a latency of two cycles, which is honored by the distance between the read stage of `gpr` and the write stage of `hi/lo`. In contrast, the dataflow between read and write of `hi/lo` involves only an addition, which has an estimated propagation delay that does not exceed one instruction cycle. These registers are therefore read and written in subsequent stages, to avoid forwarding.

8.5.8.3 Considering target frequency

The scheduler also considers the user supplied target frequency. This can clearly be seen by comparing the pipeline structure that has been discussed so far (Figure 8.19a) with the structure that is generated for a target frequency of 100 MHz (Figure 8.19b). For the lower target frequency, the multiplication is not pipelined and memory address computations are estimated to require one stage only. Besides, stages 1 and 2 are merged to a "decode and read" stage. As a result the pipeline length is reduced by 2 stages to a 5-stage pipeline.

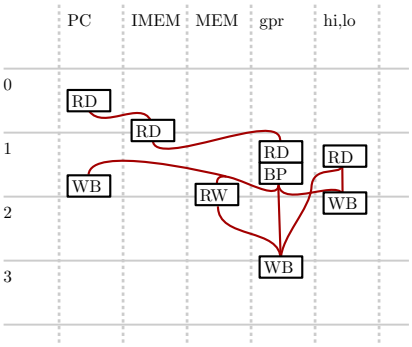
For now, this is the minimal pipeline length due to *synchronous memory accesses* and *bypass-relaxing*. The memory accesses account for two stages and relaxing for one stage. Together with instruction fetch and write back of a baseline pipeline, this adds to 5 stages. The instruction and data memory are considered to be synchronous, i.e. they have a latency of one cycle. If the address of the memory is set in cycle i, the result becomes available in cycle $i + 1$. Synchronous memory can be imagined to contain one row of pipeline registers, which account for the latency. Memory accesses are therefore placed on the border of pipeline stages, as shown in Figure 8.19.

As mentioned, one pipeline stage is introduced by relaxing of `gpr`'s bypasses. Relaxing eliminates bypass-carried critical paths, by moving dataflow into later pipeline stages. For instance, the access on `MEM` and its address computation is moved into stage 2.

8.5.8.4 Reducing pipeline length

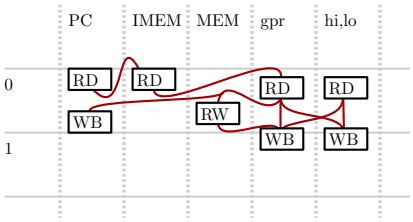
The pipeline length can be reduced by deactivating bypass-relaxing. The resulting 4-stage pipeline in shown Figure 8.20a. The access to the data memory and the write stage of `hi/lo` move up by one stage. As a result, all instructions have a latency of zero, i.e. the pipeline is never stalled during execution. Latencies are discussed later in Section 8.5.9. A major disadvantage of this simple pipeline is a longer critical path, which leads through the bypass. Synthesis shows, that the 4-stage pipeline yields a frequency of only 260 MHz, whereas the relaxed 5-stage pipeline exceeds 350 MHz. Bypass-relaxing is therefore configurable and can be controlled by the user to tailor the processor to its area of application.

(a) `MIPS_F100_nRX` processor




```
--- CONFIG ---           --- SCHEDULE ---
TargetFreq  := 100MHz    IMEM RD=0 WB=# BP=#
RelaxForward:= no        gpr  RD=1 WB=3 BP=1
MEMLatency  := 1         hi   RD=1 WB=2 BP=#
                         lo   RD=1 WB=2 BP=#
                         pc   RD=0 WB=1 BP=#
                         mem  RD=1 WB=2 BP=#
```

(b) `MIPS_F100_nRX_ASYNC` processor




```
--- CONFIG ---           --- SCHEDULE ---
TargetFreq  := 100MHz    IMEM RD=0 WB=# BP=#
RelaxForward:= yes       gpr  RD=0 WB=1 BP=#
MEMLatency  := 0         hi   RD=0 WB=1 BP=#
                         lo   RD=0 WB=1 BP=#
                         pc   RD=0 WB=0 BP=#
                         mem  RD=0 WB=1 BP=#
```

Figure 8.20: Structure of generated MIPS pipeline (a) without bypass-relaxing and (b) with asynchronous memory.

8.5.8.5 A non-pipelined implementation

The processor generator can also produce a non-pipelined processor. Therefore, the length of the pipeline is further reduced, by defining memories to be *asynchronous* instead of *synchronous*. Technically, this is achieved by setting the memory's latency in the delay library from 1 to 0. This change is a matter of seconds and does not require any modifications in the generator. The resulting pipeline is shown in Figure 8.20b. It consists of two stages and does not include any forwarding. In general, a two-stage pipeline is equivalent to a *non-pipelined* implementation. It should be noted, that the generated processor has not been synthesized, as the underlying memories are synchronous.

8.5.8.6 Comparison

The generated MIPS pipelines are similar to a manually designed pipelines. The MIPS architecture was developed in 1981 by a team around John L. Hennessy. It is described in the book *Computer architecture* [24] by Hennessy and Patterson. The authors present an exemplary MIPS pipeline, which is shown in Figure 8.21b. To evaluate the quality of the processor generator, this pipeline is compared to the generated MIPS pipeline in Figure 8.21a. For a target frequency of 400 MHz (`MIPS_F400`), the generator derives the same pipeline structure, that is proposed by Hennessy and Patterson, namely `IF,DC/RD,EX,MA,WB`. Both architectures apply forwarding only on the general purpose registers and have a load latency of one cycle, which is interlocked. Both pipelines perform branches

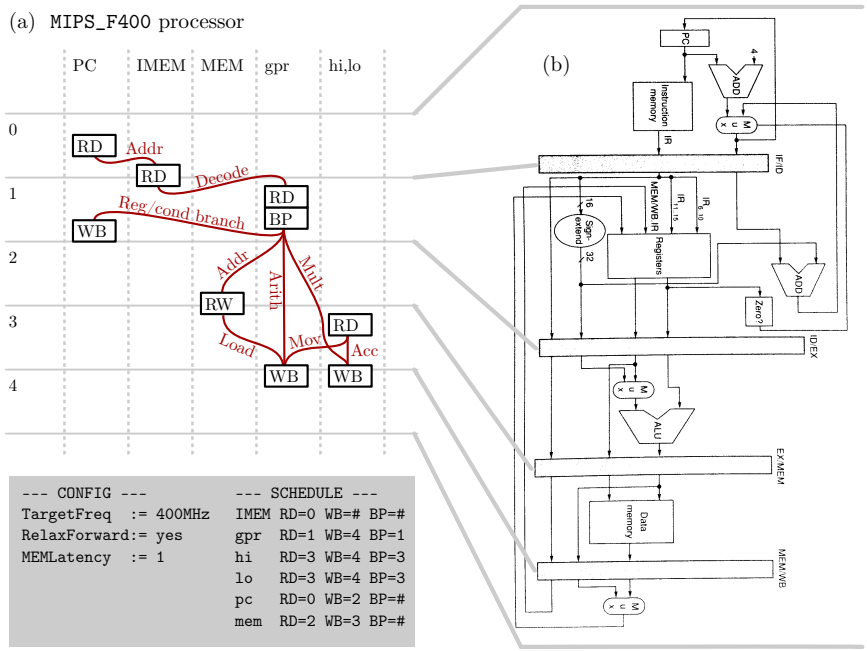


Figure 8.21: Comparison of (a) generated MIPS pipeline and (b) MIPS pipeline, as proposed in the book *Computer Architecture* [24].

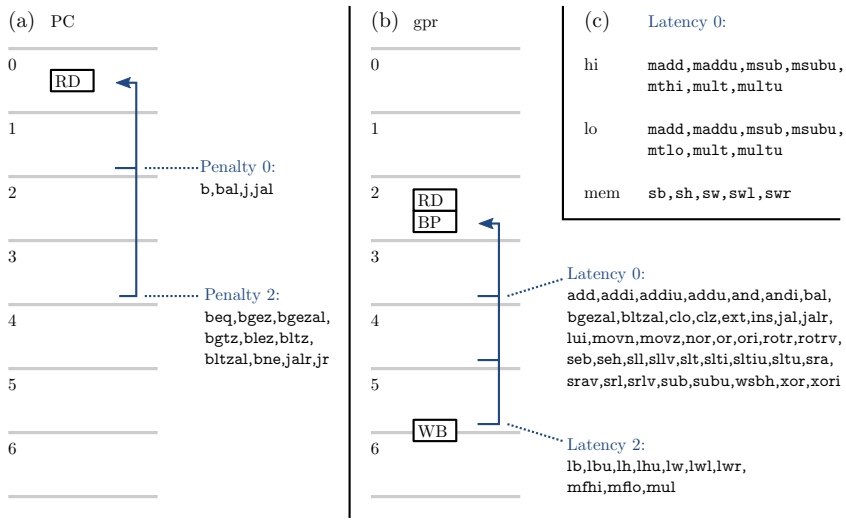


Figure 8.22: (a) Branch penalties and (b,c) instruction latencies for the generated `MIPS_F600` processor.

in stage 2. However, the generated MIPS processor does not implement branch delay-slots and triggers unconditional branches in stage 1 (Section 8.5.9). Note, that the generated multiplier is pipelined and utilizes the EX and MA stage. In contrast, the generated `MIPS_F100` pipeline in Figure 8.19b uses a non-pipelined multiplier.

8.5.9 Latencies and penalties

Instructions have a low latency, which increases with the user supplied target frequency. The same holds for penalties of mispredicted branches. Figure 8.22 shows the latencies and penalties of a `MIPS_F600` processor. Although, this processor has a long 7-stage pipeline, most instructions have zero-latency, i.e. can be executed consecutively without causing any pipeline stalls. Write accesses to `hi`, `lo` and `mem` always have a latency of zero (Figure 8.22c), due to the location of read and write stages. Data-hazards on the general purpose register file are mostly resolved by forwarding (blue arrow). Only the load instructions and the "move from hi/lo" instructions (`mfhi`, `mflo`) have non-zero latencies. The latency of the latter can be attributed to the pipelined multiplication. Effectively, the latency of multiplication instructions is transferred to `mfhi` and `mflo` via the `hi`/`lo` registers. As a result, data-dependent multiply-accumulate instructions can be issued in every cycle (e.g. in the context of signal processing applications), without causing any stalls. The latency of load instructions results from the synchronous data memory and address computations. Summing up, most data-hazards are resolved by forwarding. Only few "critical" instructions involve interlocking and have a non-zero latency. The majority of instructions has zero-

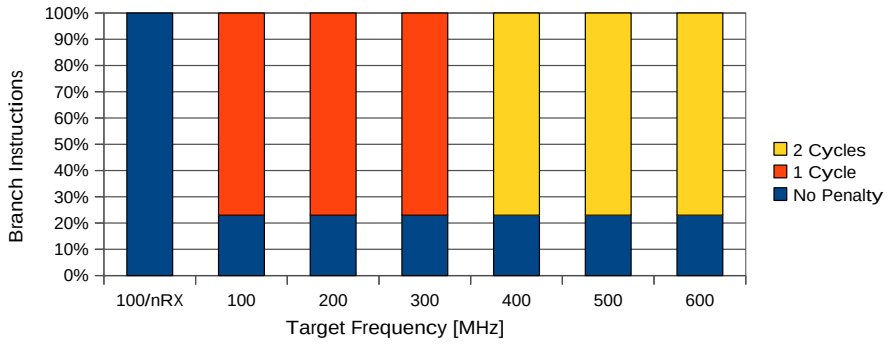


Figure 8.23: Effect of target frequency on branch penalties for MIPS.

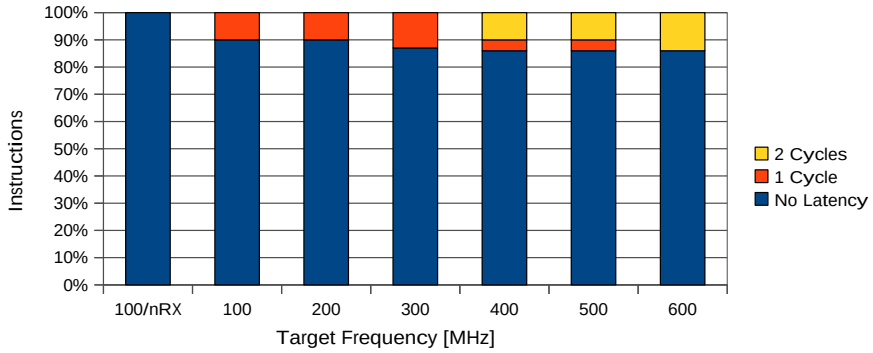


Figure 8.24: Effect of target frequency on instruction latencies for MIPS.

latency and can tightly be scheduled without causing any stalls during execution.

The penalty for mispredicted branches depends on the semantics of branch instructions. Relative branches that are executed unconditionally have a penalty of zero. Executing such instructions does not involve any speculative execution and instruction-canceling. Misprediction of other branches (register branches and conditional branches) causes a penalty of 2 cycles. These branches rely on results that are computed in stage 3. The instructions in stage 1 and 2 are therefore executed speculatively and may be canceled, which implies a penalty of 2.

8.5.9.1 Effect of configuration

The latencies and penalties of an instructions increase with target frequency. They can be reduced by deactivating bypass-relaxing. Figure 8.24 shows the percentage of instructions with a latency of 0, 1 and 2 cycles for different configurations. The same is done for branch instructions and their penalties in Figure 8.23. For a target frequency of 100 MHz and suppression of bypass-

relaxing (`100/nRX`), all instructions have a latency of 0 and all branches have a penalty of 0. This means, that the processor does not include any interlocking. With bypass-relaxing activated, 77% of branches have a penalty of one cycle for `MIPS_100` to `MIPS_F300` and a penalty of two cycles for `MIPS_400` to `MIPS_F600`. The remaining 22% of branches are unconditional-relative and have a penalty of 0.

With a rising target frequency, the percentage of zero-latency instructions drops from 90% to 86%. Basically, the latency of load and "move from hi/lo" instructions increases, till it reaches 2 cycles. Nevertheless, most processor instructions have a latency of zero. Their result is directly available to subsequent instructions via forwarding, without a need for pipeline stalls.

8.5.9.2 Instruction scheduling

All information on branch penalties and instruction latencies presented so far has been emitted by the processor generator. It is validated in Section 8.5.10 by simulating the MIPS processor on RTL level. The generator solely derives latencies and penalties from instruction semantics. In particular there is no need for any manual analysis or specification of further aspects of instructions.

The derived information is not only interesting for evaluation, but may also be passed to a compiler generator. The scheduler of the compiler can then consider latencies, such that stalls are avoided. In addition the delay information that is specified in ViDL can be passed to the scheduler. A scheduler that considers both, delays and latencies is outlined in [12].

8.5.10 Resolution of hazards

So far, the static pipeline structure and resulting properties of instructions by means of latencies have been discussed. This section is primarily intended to demonstrate the resolution of data and control hazards in the generated processor. Therefore, small test programs are executed on the processor. An image of each program is loaded into the instruction memory of the simulated processor. The entire processor core (VHDL code) is then simulated in ModelSim on RTL level. This includes the pipeline, forwarding, interlocking and pipeline registers. In contrast, these microarchitectural aspects are not considered by the generated simulator, which focuses on the instruction set for sake of efficiency.

This section also serves as a proof of concept to show that the pipeline *works correctly* and that it has a *high throughput* of instructions. The throughput is for instance not limited by unnecessary pipeline stalls or excessive instruction canceling. The executed programs test error-prone aspects of the processor, such as forwarding, interlocking, speculative execution and canceling.

Before the execution of programs is discussed, the visualization of waveforms in ModelSim is explained. This includes a description of relevant signals of the generated processors. Afterwards, the resolution of control-hazards (Section 8.5.10.2) and data-hazards (Section 8.5.10.3) is demonstrated. The resolution of data hazards is also compared among different processors that have been

targeted at different frequencies (Section 8.5.10.4). But first, a short introduction to waveforms is given in the next section.

8.5.10.1 Waveforms in ModelSim

ModelSim is a tool to simulate hardware descriptions, such as the VHDL code of a processor. The state of signals over time is visualized in ModelSim by so called waveforms. This section explains how to read such waveforms and describes the meaning of relevant processor signals. For the explanations, Figure 8.25 is regarded in the following. It shows the assembly code of a test program on the left and a screenshot of ModelSim on the right. The screenshot shows the waveforms of selected processor signals. Time advances from left to right. A timeline is shown at the bottom, together with annotated cycle numbers (yellow). Interesting parts of the waveforms have been annotated (yellow) and will be discussed in the next sections. Waveforms have hexadecimal values or the special value X, meaning "unknown". The latter shows up, when a register is read, before it has been written. These values can be ignored, as they do not affect execution of correct programs.

The program counter signal (`pc/next`) refers to the instruction address in stage 0 and matches the instruction address (IA), which is shown on the left of program code. During execution of linear code, the program counter is increased by 4 in each cycle. A value of 1 for the `pc/trigger/Sx` signal means, that a branch is triggered by the instruction in stage `x`. The `gpr/write0/` signals denote the address (`idx`), the value (`val`) and the write enable (`we`) of a write access to general purpose registers. The `opc/Sx` signal shows the name of the instruction that is located in stage x. These signals nicely visualize the flow of instructions through the pipeline. A value of 0 for the `nCancel/Sx` signal means, that the instruction in stage `x` has been canceled. The instruction does neither write its result back to registers nor does it bypass the result to subsequently executed instructions. Cancel signals are not displayed for all stages, as some have been optimized out by the processor generator.

The assembly code on the left of Figure 8.25 uses a conventional notation, as understood by the gnu assembler (`gas`). Two things should be noted: First, `li` (load immediate) is a pseudo instruction, that is translated into `addiu` and second, the registers `$t0` to `$t3` are actually aliases for `gpr[8]` through `gpr[11]`.

8.5.10.2 Control hazards

Figure 8.25 shows the execution of a test program, which contains conditional (`beq`) and unconditional branches (`b`,`j`). In cycle 2, the branch `b` at address 34_h is located in stage 1 and triggers a branch to `x1` at address 40_h. No instruction needs to be canceled. In contrast, the conditional branch `beq` at address 44_h triggers a branch to x2 in cycles 6 and stage 3. The speculatively executed instructions at address 48_h and $4\mathrm{C}_\mathrm{h}$ are therefore canceled in this cycle. Due to canceling, the write enable signal of `gpr` is disabled in cycle 8 and 9. The conditional branch at address $5\mathrm{C}_\mathrm{h}$ is not taken. The speculatively issued instructions

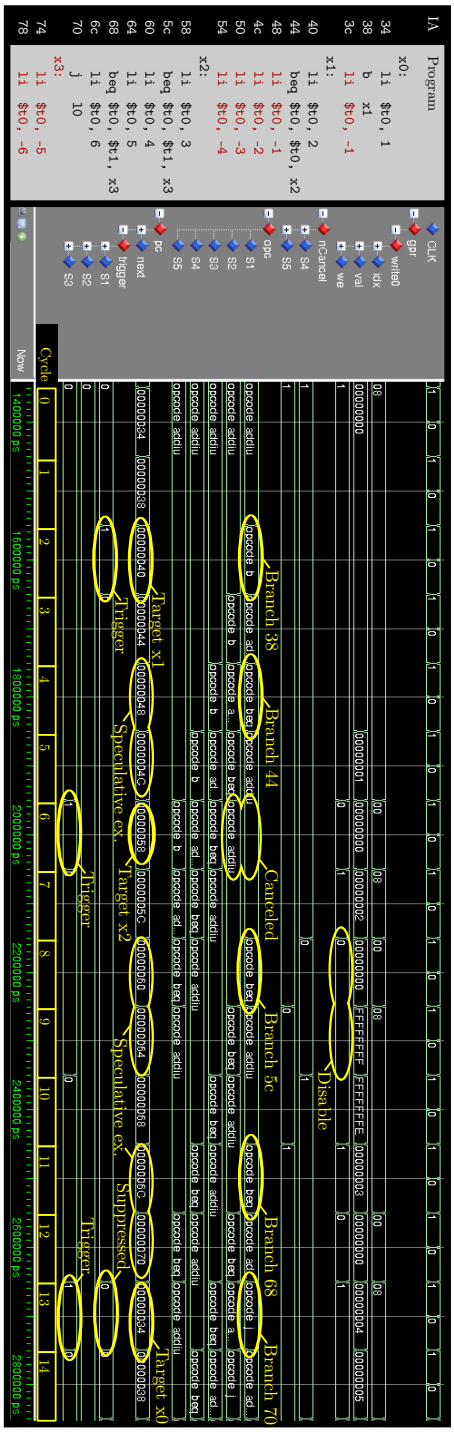


Figure 8.25: Speculative execution in the generated `MIPS_F600` processor.

in cycle 8 and 9 are therefore not canceled and take effect. The last two branche instructions at address 68_h and 70_h are in an interesting constellation. The second instruction is executed speculatively. Both branch instructions trigger a branch in cycle 13. However, the second branch is suppressed, as it is canceled by the first branch. Execution therefore continues at address 34_h, as expected.

To sum up, the example has demonstrated that speculative execution and canceling works correctly. This also holds true for conflicts between speculatively executed branches. In case of misprediction, only few instructions need to be canceled, which results in an efficient execution.

8.5.10.3 Data hazards

Figure 8.26 shows the execution of a small program, which demonstrates forwarding and interlocking (stalling). It contains applications of the `mfhi`, `mflo` and `lw` instructions, which have a latency of 2 cycles for the regarded `MIPS_F600` processor. The other instructions, including multiply and multiply-accumulate (MAC) have zero-latency. In cycle 3, the multiplication at address 24_h reads its source operands `gpr[8],gpr[9]` in stage 2. The values are taken from the bypass, as the results (2 and 3) of the previous instructions (`addiu` and `addi`) have not yet been written back to `gpr[8]` and `gpr[9]`. The register still contain the outdated values 0 and -1 ($FFFFFFFF_h$). Besides, forwarding on `gpr` is utilized in cycles 2, 4, 9, 10, 11, and 14. In cycle 6, the result of the multiplication appears in `hi` and `lo`. The subsequent zero-latency MAC instruction at address 28_h is executed without any need for forwarding or interlocking. Its result appears in `hi/lo` in the following cycle.

In contrast, the `mfhi` and `mflo` instructions cause a stall of the data-dependent `add` instruction in stage 2 at cycles 7 and 8. As a result, stages 0 and 1 are stalled as well and bubbles are inserted in stage 3 at cycle 8 and 9. The bubble appears as a canceled `add` instruction, which does not take effect. In the same way, the `addi` instruction at address 40_h which depends on the loaded value from `lw` is stalled in cycles 12 and 13. The result of the last instruction of the program is written back in cycle 17.

The example has shown that forwarding and interlocking is correctly applied to solve data-hazards. Interlocking is only applied if necessary, to maximize instruction throughput. Considering pipelining, the execution of these 10 instructions takes 14 cycles. The average number of cycles per instructions (CPI) is therefore 1.4.

8.5.10.4 Comparison of microarchitectures

Figure 8.27 shows the execution of the program from Figure 8.26 on 7 different MIPS processors, which have been generated from the same specification. Next to each simulation, the name of the processor, its configuration, the derived latencies, and the estimated clock frequency are shown. Besides, the average CPI and the resulting number of “million-instructions-per-seconds” (Mips) for

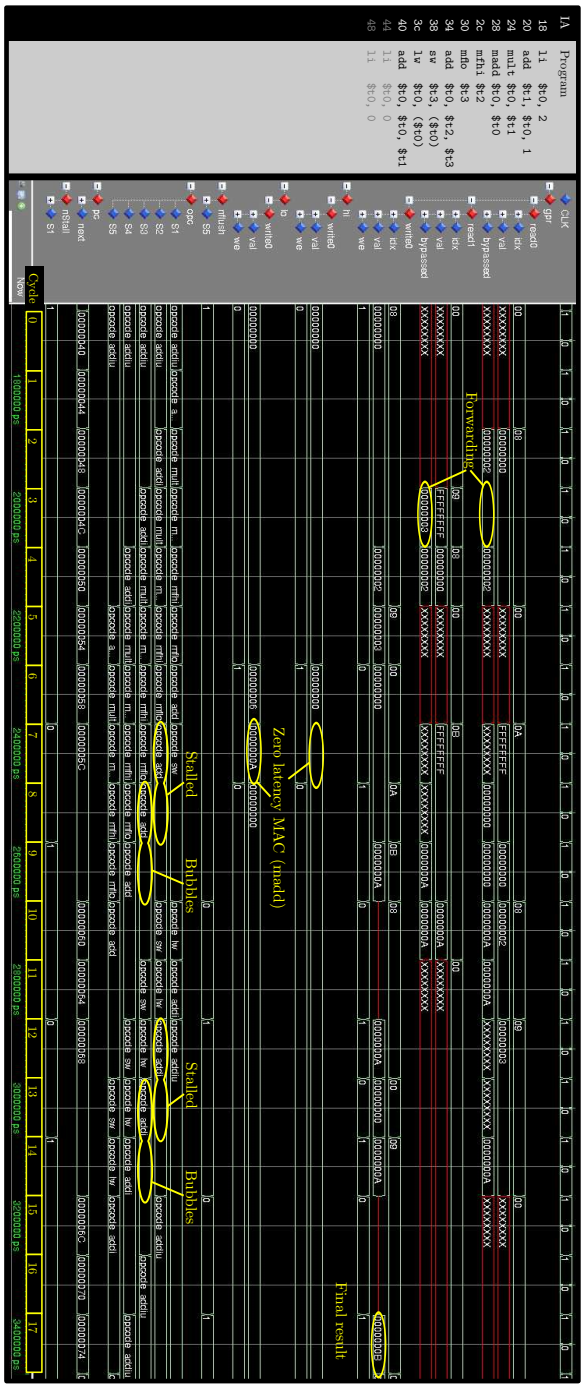


Figure 8.26: Resolution of data hazards by forwarding and interlocking in the generated `MIPS_F600` processor.

Processor	MIPS_F600	MIPS_F400	MIPS_F300	MIPS_F100	MIPS_F100_nRx
TargetFreq	600 MHz	400 MHz	300 MHz	100 MHz	100 MHz
RelaxForward	yes	yes	yes	yes	no
Stages	7	7	5	5	4
LW latency	2	2	1	1	0
MFxx latency	2	1	1	0	0
SynthFreq	660 MHz	650 MHz	500 MHz	360 MHz	260 MHz
CPI	1.4	1.3	1.2	1.1	1.0
MIPS	470	500	420	330	260

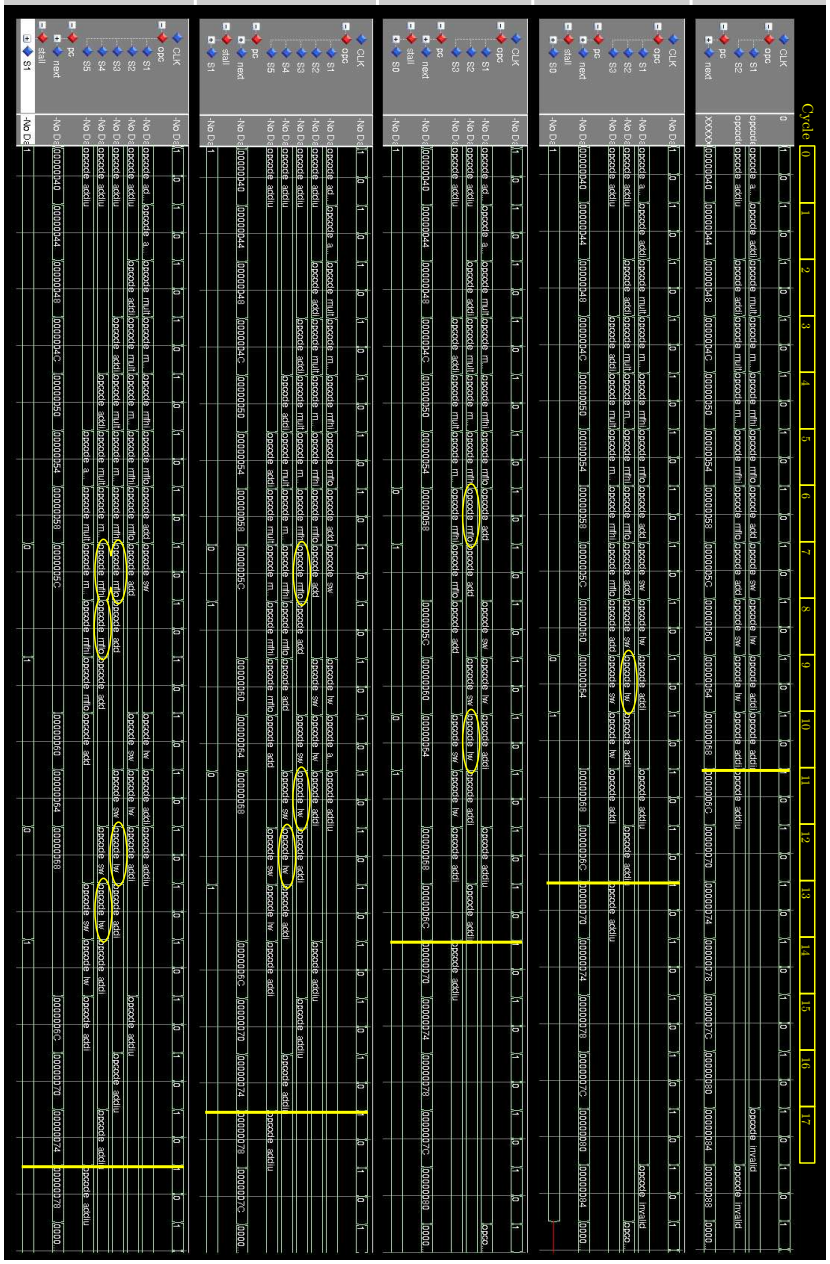


Figure 8.27: Execution of the same program on different generated MIPS processors and its dynamic effects.

this program is shown. Instructions that cause data hazards are highlighted in the waveforms, as well as the end of program execution.

The `mips_F100_nRX` processor does not involve any interlocking. The CPI is therefore 1.0 and the number of Mips corresponds[7] to the clock frequency from synthesis (260 MHz). With an increasing target frequency, the latency of `lw`, `mfhi` and `mflo` rises, which results in an increasing number of stalls during execution. The CPI therefore rises as well. For `mips_F100_nRX` through `mips_F400`, the clock frequency rises stronger than the CPI. For `mips_F600`, the increase in clock frequency does not compensate the increase in CPI, which results in a reduced number of Mips. Assuming, that an efficient processor is required and that this program is representative for its area of application, a developer will likely select the `mips_F400` processor, as it has the highest throughput of instructions.

8.5.11 Generating waveform definitions for ModelSim

The processor generator includes a well working extension to generate waveform definitions for ModelSim. In ModelSim, waveforms are used to observe the behavior of an integrated circuit. To test and evaluate a processor, a set of significant waveforms needs to be defined. This includes meaningful grouping of waveforms, as well as a concise and schematic naming.

Defining and maintaining dozens of signals for each generated processor is quite tedious. The processor generator has therefore been extended to generate these waveform definitions. The definitions are correct and consistent with the generated VHDL code. An additional register port will for instance result in further waveforms to be generated. The waveforms are grouped and use a unified naming scheme. This allows for simple navigation in the waveforms of different processors. The extension of the generator has been quite complex. Most signal identifiers need to be assigned in early phases of generation. This association then has to survive subsequent transformations and optimizations in the generator. The task is related to the handling of origin information (Section 7.2.3).

During development, a bug has been found in ModelSim 6.1. If two signals (of different groups) share the same name, the display of waveforms is unpredictable. The generator circumvents this problem by adding a varying amount of white space after each name, such that they become literally unique. Their display on the other hand remains unaffected.

All waveforms shown in previous examples have been defined by the processor generator. Figure 8.28 shows the number of waveforms for different processors. The number of signals clearly increases with the complexity of the instruction set and the length of the generated pipeline. Thanks to the generator extension, a processor can immediately be simulated and observed. The waveforms are processor specific and consistent to the VHDL code. No additional information needs to be supplied by the developer. Changes to the ViDL

[7] Since the processor core is to be evaluated, stalls from the memory subsystem are not considered.

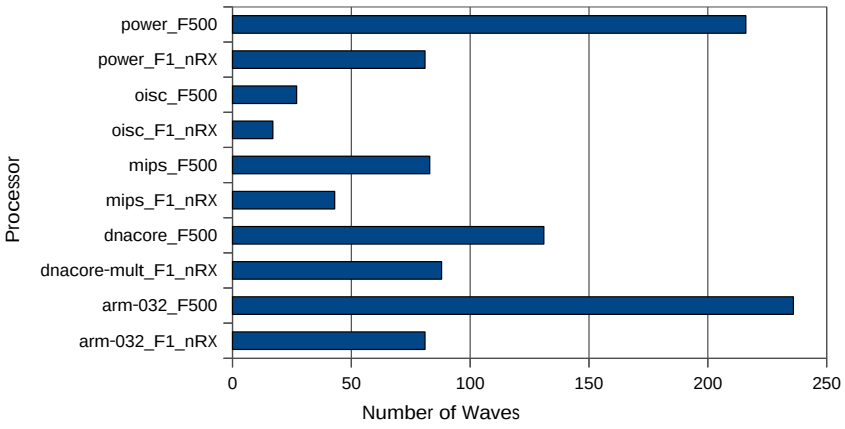


Figure 8.28: Number of defined waveforms for different processors.

specification or the microarchitectural configuration directly affect waveform definitions.

8.6 DNACore — A case study on ISE

So far, distinct aspects of the language ViDL, its generators and their products have been evaluated. This final section presents an all-embracing application of the system in a typical design scenario, to demonstrate its fitness for practice. Therefore, the existing MIPS specification is enriched by an instruction set extension (ISE), which accelerates the execution of the Smith-Waterman algorithm. The resulting instruction set is called *DNACore*. The Smith-Waterman algorithm [46] is used in bioinformatics to identify similar molecular subsequences in DNA, RNA or proteins. A generated DNACore processor compute 2.6 GCUPS or 57 GCUPS/W. With respect to energy efficiency, it outperforms desktop-CPU based approaches by a factor of 100 [56].

8.6.1 Development process

To develop the extension, the Smith-Waterman algorithm has been *analyzed* and typical *use cases* in bioinformatics have been gathered (Section 8.6.2). The resulting extension exploits data parallelism for acceleration and data locality to reduce memory accesses (Section 8.6.3). It has been *specified in ViDL* and added to an existing MIPS specification, yielding the DNACore instruction set (Section 8.6.4).

The *dynamic behavior* of DNACore processors has been inspected, by regarding latency and pipeline information, which is supplied by the processor generator. For evaluation, an implementation of the Smith-Waterman algorithm has been *programmed*, which utilizes the ISE. The extension has been

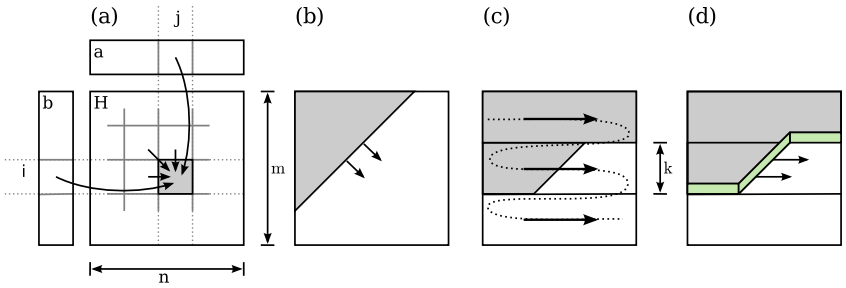


Figure 8.29: Data dependences and computation strategies for H.

tested by executing the program on the generated simulator. After testing, the same program has been executed on the generated processors in ModelSim, to *validate* the predicted dynamic behavior (Section 8.6.5). Finally, the generated DNACore processors have been *synthesized* for a 65nm STMicroelectronics low power technology. Their *physical characteristics* have been presented in Section 8.5.3, along with other processors.

8.6.2 Algorithm

The Smith-Waterman algorithm [46] uses dynamic programming to find the optimal local alignment of two sequences with respect to a scoring function w. The algorithm is computation intensive, as its runtime is quadratic in the length n of the compared sequences. In the following, the input of the Smith-Waterman algorithm is assumed to be a pair of two sequences a, b of length n and m in an alphabet Σ, where $n \geq m$ holds.

$$a = (a_1, \ldots, a_n) \in \Sigma^n$$
$$b = (b_1, \ldots, b_m) \in \Sigma^m$$

To find the optimal local alignment, the algorithm computes a matrix H of size $m \times n$, which is defined by the following recurrence

$$H(i,j) = \begin{cases} 0 & \text{for } i = 0 \vee j = 0 \\ \max \left\{ \begin{array}{l} 0 \\ H(i-1, j-1) + w(a_i, b_j) \\ H(i-1, j) + w(a_i, '-') \\ H(j-1) + w('-', b_j) \end{array} \right\} & \text{for } 1 \leq i \leq m \wedge 1 \leq j \leq n \end{cases}$$

The data dependences within the matrix H are illustrated in Figure 8.29a. Using m processing elements (PEs), the matrix can be computed in $n + m$ steps, as shown in Figure 8.29b. As only few elements are computed at the beginning and at the end, utilization of PEs is low (≈ 0.5). In general , the

number of computation steps and the utilization is given by

$$\text{Steps} = \frac{(n+k-1)m}{k}$$
$$\text{Util} = \frac{n}{n+k-1}$$

To yield a good trade-off between speed and utilization, the proposed instruction set extension uses $k \ll m$ PEs to compute k rows of the matrix in parallel, as shown in Figure 8.29c. For typical values like $n = m = 512$ and $k = 4$, this yields a high utilization of 0.99. During 100 computation steps, the PEs compute 397 elements of the matrix.

If only the alignment score is to be computed, memory accesses to the matrix H can be reduced. Figure 8.29d shows, which parts of H are alive (green), i.e. are going to be read by pending computations. To compute the score, it is sufficient to store only these n alive elements of the matrix.

8.6.3 Instruction set extension

The instruction set extension consists of 7 instructions, which update multiple cells of H in parallel. It also defines a set of dedicated processor registers, to reduce utilization of the main memory. Besides, memory accesses are reduced by packing multiple symbols into one data word. In the following it is assumed, that $l = 4$ adjacent 8-bit-values of a, b and H are packed into one 32-bit word. This way, 4 symbols are transferred by one memory access.

The set of internal registers is shown in Figure 8.30a for $k = 4$ PEs. Data is transferred between memory (gray) and 32-bit registers (blue), which hold 4 8-bit values. The purple registers represent an opaque internal state of the ISE. The 4 PEs are indicated by black dots. Figure 8.30b illustrates the dataflow (arrows) between registers. The registers that are accessed by PE1 are highlighted. Each PE updates one cell of H and a register of M which contains the maximum score.

A Smith-Waterman instruction integrates all PEs and the shift on registers A and H. After $l = 4$ executions of a Smith-Waterman instruction, register A and the top of register H are reloaded with data from memory. The bottom of register H then contains 4 matrix elements, which are written back to memory. In total, this makes 3 memory operations for 4 cycles of execution. Fortunately, these memory accesses can be performed in parallel, by folding them into Smith-Waterman instructions. Further instructions are specified to load register B, to get the score from M, and to initialize internal registers. In total, the instruction set extension consists of 7 instructions.

8.6.4 Specification in ViDL

The DNACore instruction set has been specified by extending the existing MIPS specification. The Smith-Waterman instructions have been embedded into an unused region of the instruction space, which is intended for extensions. Only

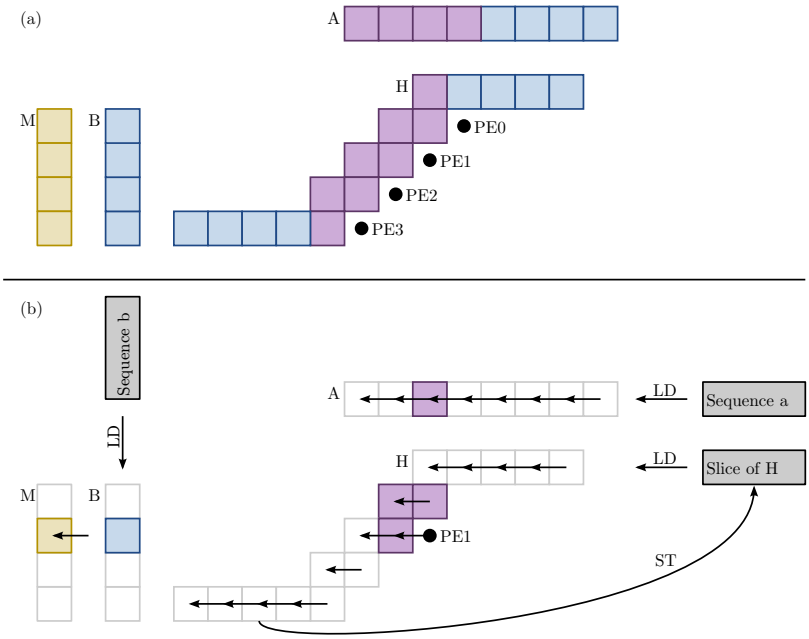


Figure 8.30: (a) Internal registers of the Smith-Waterman ISE and (b) their data dependences.

the 7 instructions and the internal registers had to be specified. The complete extension consists of less than 100 lines of ViDL code and took about half a day. This includes the generation of a highly optimized simulator and several processors. The total development time (about two weeks) was dominated by the analysis of the Smith-Waterman algorithm, its dependencies and the development of an ISE idea.

Actually, two DNACore instruction sets have been defined. The first includes all instructions of MIPS and the new instructions. The resulting processors are therefore 100% compatible to the generated MIPS processors. The second instruction set is equal, but does not include the multiplication instructions, as they are not required to execute the Smith-Waterman algorithm. Since the generated processors of this instruction set do not include the 64-bit multiplier, they are smaller, faster and require less power. Unless noted otherwise, this instruction set is regarded in the following.

8.6.5 Dynamic behavior of processor

The Smith-Waterman algorithm is executed efficiently on the generated DNA-Core processor. In particular, execution is not slowed down by memory bottle necks or data-hazards. All instructions of the extension have zero-latency, i.e. they do not cause any stalls. As a result, the PEs are highly utilized.

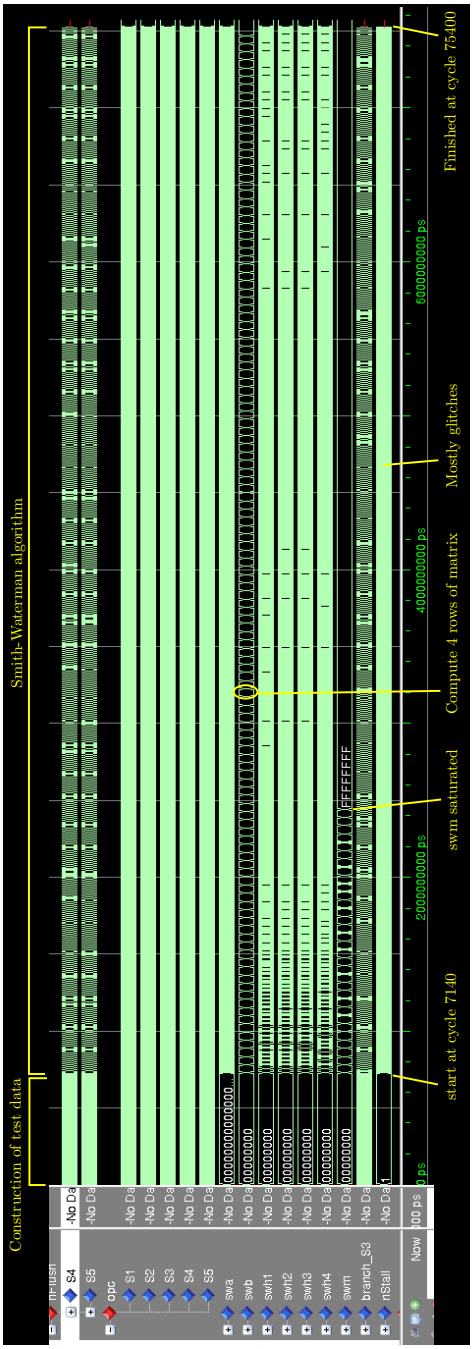


Figure 8.31: Execution of the Smith-Waterman algorithm on the generated DNACore processor.

For the following evaluation, the alignment of two sequences is considered, where each sequence has a length of 512 symbols. The alphabet Σ has a size of 256 symbols. The actual value of symbols does not have an effect on execution. In particular, control flow and memory accesses are not affected. To allow for simple reasoning, two identical sequences are regarded in the following.

The execution of the entire algorithm on the generated processor is shown in Figure 8.31. During the first 7140 clock cycles, input data is generated. This part is only required for testing and can therefore be ignored. The actual Smith-Waterman algorithm starts in clock cycle 7140 and ends in cycle 75400. During these 68260 cycles, 262144 cells of the H-matrix are computed. On average, this makes 3.84 cell updates per cycle. The 4 PEs are thereby utilized by 96%. Note, that this is a practical value, including stalls, branch penalties and memory accesses.

As the algorithm processes 4 rows in parallel, the algorithm scans 128 times over the matrix. Accordingly, the `swb` register is loaded 128 times, as one can see in the figure. The `nStall` signal seems to change heavily, however these are only glitches from simulation. Actually, the pipeline is hardly ever stalled. The score register `swm` is a vector of 4 8-bit values, one for each processing element. The score is stored separately for each PE, to reduce the critical path. The final score is computed once at the end of the entire algorithm. Since the input sequences are equal, the score rises rapidly, whenever the diagonal of H is processed. It is finally saturated, as shown in the Figure.

8.6.6 Results and remarks

The synthesized DNACore reaches a worst case frequency of 680 MHz at an estimated power consumption of 45mW and a chip area of 0.17mm^2. Since the processor executes 3.84 cell updates per cycle, this makes 2.6 GCUPS (giga cell updates per second). This value lies in the order of magnitude of related approaches. Since the DNACore is very energy efficient (58 GCUPS/Watt) and small, it may be instantiated multiple times to process many alignments in parallel. As sequence alignment is a data-parallel problem, throughput should scale well with the number of processors. Alternatively, the number of PEs (k) may be increased.

Although the DNACore includes special support for sequence alignment, it is still a general purpose processor. It can be programmed like any other MIPS processor. This is expected to be beneficial in combination with heuristic algorithms that are based on Smith-Waterman, but compute only certain parts of the H-matrix. These algorithms can be implemented using the MIPS instruction set, whereas the Smith-Waterman kernel is accelerated by the ISE.

8.7 Summary

Using ViDL and its generators, efficient simulators and processors have been generated from specifications of real world instruction sets. This includes ARM,

MIPS, Power and CoreVA. Besides, an instruction set extension for MIPS has been developed and implemented. A simulator and more than 5 processors with different microarchitectures have been generated for each instruction set.

Language

For evaluation, 4 real world instruction sets (ARM, MIPS, Power and CoreVA) and 2 academic instruction sets (OISC and SRC) have been specified using ViDL's powerful and orthogonal concepts, such as epsilon logic and delays. This includes delayed branch instructions, exceptionally wide arithmetic and auto-incrementing addressing-modes. Instruction sets have rapidly been specified, due to a high level of abstraction and effective concepts for reuse. The MIPS instruction set has been specified in one day and its DNACore extension in half a day.

Specifications are concise, as bit-widths of operations and microarchitectural aspects, such as register ports and pipeline stages are not specified. The semantics of most MIPS instructions require only one line of specification. Specifications are clear and reliable, due to strict separation of state, transfer, and I/O. Side effects and dependencies between instruction specifications are excluded by design. Respective errors have not been encountered during development. Instead, an ambiguity in the pseudo code of the ARM manual has been detected by the generator's type analysis. In detail, the width of the rotate operation of the `MSR` instruction is not well defined in the manual and lacks a sign extension.

ViDL is easy to learn, as it does not require knowledge on C, VHDL or processor microarchitectures. It has been used by inexperienced users to develop simulators and processors of Power and CoreVA. ViDL allows for rapid DSE and ISE, due to loose coupling of specifications (e.g. independent instructions). Instructions have rapidly been removed (multiplication) or added (DNACore ISE), without touching any other part of the specification. The width of the ARM ISA has been explored, by redefining the width of general purpose registers.

ViDL provides a high degree of reuse and maintainability, thanks to functional concepts and a library of transfer primitives. Addressing modes and the SIMD concept have been encapsulated using functions and functionals. A set of approximately 50 transfer primitives has been specified, which is reused among all specifications. Products that are generated from a ViDL specification are guaranteed to be consistent, due to strict and high level abstraction. The simulator and all generated processors are semantically equivalent. In contrast, other approaches rely on redundant specification of C and VHDL, which may result in inconsistent simulators and processors

Simulator

High-performance simulators have been generated for all instruction sets. They execute 60 Mips on average and 140 Mips peak on a 3 GHz Intel workstation. Delayed branches and bit-precise write-enabling is efficiently simulated. The same holds for ultra-wide instruction sets ($\approx$ 30 Mips for 256-bit arithmetic), as the

generator breaks wide arithmetic down to efficient C-integer arithmetic. This task, as well as masking of short bit-strings is shifted from the ISA developer to the generator.

The generator includes a series of optimizations, which significantly increase simulation speed. For instance, lazy evaluation accelerates simulation by 100% on average. Merging of write accesses accelerates simulation by up to 90%. The implementation width of a simulator can be selected at generation time. Tailoring the width of a simulator to the host system accelerates simulation by up to 125%.

Processors

For each instruction set (ARM, MIPS, Power and DNACore), efficient processors have been generated and synthesized for a 65 nm STMicroelectronics standard cell technology and worst case conditions. Their clock frequency lies in the range of 600 MHz to 1.1 GHz, for a deeply pipelined microarchitecture. For less pipelined microarchitectures, clock frequency lies in the range of 350 MHz to 780 MHz. For this microarchitecture, chip area lies in the range of 0.03 mm^2 to 0.32 mm^2 and power consumption in the range of 15 mW to 65 mW. As shown for CoreVA, the clock frequency of generate processors lies in the range of hand-crafted VHDL code. Estimates for area requirement and power consumption are larger by a factor of 3. First examinations of the intermediate representation have revealed a high potential for hardware sharing in the generator. Currently, only a very simple hardware sharing method has been implemented.

Processor generation is guided by a user supplied target frequency. This parameter significantly affects physical characteristics of the processor, as well as the dynamic behavior of its instructions. It has been used to generate small and energy efficient processors, as well as deeply pipelined high-performance processors from the same specification. Based on this parameter and instruction semantics, all aspects of the microarchitecture are derived automatically. This includes forwarding, interlocking and speculative execution. The resulting pipeline control is correctly implemented by the generator, such that instruction semantics are preserved. For a target frequency of 400 MHz, the microarchitecture of the generated MIPS processor matches the pipeline structure that is proposed by Hennessy and Patterson.

DNACore

As an all embracing proof of concept, the existing MIPS instruction set has been enriched by a SIMD instruction set extension for the Smith-Waterman algorithm. The extension, has been developed in the context of this thesis, consists of 7 instructions, 7 internal registers, and has been specified in half a day. Synthesis has estimated a clock frequency of 680 MHz at a power consumption of 45 mW for the generated processor. According to hardware simulation, the 4 processing elements of the SIMD extension are utilized by 96%, which results in a performance of 2.6 GCUPS or 58 GCUPS/W.

Chapter 9

Conclusion

This thesis has shown how efficient processors can be generated from specifications of instruction sets and only from those specifications. All microarchitectural aspects of processors are automatically derived from instruction semantics, without exception. This principle sets the presented approach apart from related work. A simulator and very different processors with well balanced physical and dynamic characteristics are generated from the very same instruction set specification, guided by a user supplied target frequency only. Processor implementations range from non-pipeline processors, via stall and penalty free implementations to 6 stage pipelines, which solve hazards by forwarding, interlocking and canceling. The generated simulator and all generated processors are guaranteed to be consistent to the instruction set specification. There is no way, a ViDL developer can break this consistency, neither by accident, nor by intention. The need to test equivalence of different implementations is thereby eliminated.

The methods for pipelines construction have proven to be effective for real instruction sets. The clock frequency of generated processors lies in the order of handcrafted VHDL code. Power consumption and area requirements are approximately 3 times higher, but first examinations have revealed a high potential for hardware sharing methods in the generator. The average CPI of generated processors is close to one, i.e. in the order of manually designed processors. A generated processor does therefore actually utilize its high clock frequency, to yield a high throughput of instructions. Generated simulators are up to 6 times as fast as other interpreting simulators. Ultra-wide instruction semantics (e.g. 256-bit ARM) are automatically broken down to plain C-arithmetic, resulting in fast simulation.

ViDL obeys fundamental language design guidelines and thereby allows for specifications of high-quality. Novel language concepts, such as *architectural interfaces*, *epsilon logic*, *delays* add to simplicity, reliability and expressiveness, while retaining strict abstraction from implementation level. A high degree of reuse and maintainability is enabled by functional concepts, polymorphism and a sophisticated *type system*. A respective *type inference* also checks static

semantics and reports undefined or ambiguous aspects of the specification. Reliability is further improved by the side-effect free language design and the clear separation of state and transfer.

Using the system of ViDL and generators, the design space of an instruction set can easily be explored, due to *loose coupling* in specifications and sophisticated language concepts. Instructions can be added, removed or modified independently without the need to reconsider the entire specification. The microarchitectural design space of processors can automatically be explored by repetitive generation with different target frequencies. The resulting VHDL code is ready for synthesis and simulation in ModelSim. For the latter, well structured waveform definitions are generated, to make significant processor signals immediately observable. Existing instruction sets are easily extended by application specific instructions and registers. The resulting processors are guaranteed to be backward compatible, i.e. testing of existing instructions is unnecessary. The specification of 4 major instruction sets (ARM, MIPS, Power, and CoreVA) and one instruction set extension (DNACore) demonstrates the quality of ViDL and proves the effectiveness of generation methods. The design space of instruction sets and their microarchitectural implementation is rapidly explored. The system has been adopted by new users (students) in a short time and is ready to be utilized by further developers to accelerate their design flow. I'm looking forward to its application in serious industrial environments in near future. The system is expected to bring simplicity to the development of sophisticated processors. This makes development more reliable and affordable for a wider range of applications. For instance, a variety of small and highly specialized processors can be developed within few days and integrated into a system on a chip.

Future work

The proposed system is well prepared for extensions, as ViDL uses a high level of abstraction and the generators feature a modular design. The set of generators may be extended, to produce further products, such as compiler tools. The existing generators can be refined, to improve the efficiency of produced simulators and processors. The language ViDL on the other hand may be extended, to enable the natural specification of less common aspects of instruction sets.

Compiler toolchain

A compiler toolchain consists of a compiler, an assembler, a linker and further tools for debugging and conversion. A generator that produces these tools from a ViDL specification would be a valuable extension. Although such generators are beyond the scope of this thesis, they have been considered during the design of ViDL. The specification language UPSLA has been developed at our research group and supports generation of complete toolchains, including compiler, assembler and linker. As ViDL defines instruction sets on a similar level,

it should be well suited to generate compiler tools. However, ViDL does not directly define compiler relevant aspects of instructions. A good compiler generator should derive these aspects from instruction semantics by analyses, to guarantee consistency and simplify specification. According to my experience[1] in compiler tool development, I've a quite clear idea on the implementation of such generators. I consider the development a good deal of work, but feasible and worthwhile. Alternatively, a generator may be developed, which translates a ViDL specification into an UPSLA specification. The UPSLA generators can then be applied.

Tuning of generated HDL code

As a matter of principle, ViDL strictly abstracts from all microarchitectural aspects. The goal is to develop the processor generator to the extent that generated HDL code is of the same quality as highly optimized handcrafted code.

Until then, it may be desirable to give the developer a possibility to manually fine-tune the generated HDL code. For instance, tuning commands may define the location of read and write stages for each storage. To retain abstraction in ViDL, such commands for the generator should be defined in a separate "tuning" file. To refer to instruction semantics, operations in ViDL may be annotated with identifiers, as usual for entity instances in VHDL. The identifiers could then be used in tuning commands.

Nevertheless, manual tuning significantly increases the effort for development and reduces maintainability. Extending the generator by further methods that encapsulate knowledge of processor developers is still the preferred solution. Besides, direct modification of generated VHDL code is strongly discouraged, as this will prevent future modifications of the ViDL specification and regeneration.

Close gaps in generators

In Section 8.2.5, aspects of the generators and ViDL have been mentioned, that are not supported yet. The section also outlined how these gaps can be closed. The extensions are considered a matter or programming. Closing these gaps is a prior objective, to eliminate restrictions on instruction set specifications.

Microarchitecture

Currently, the processor generator produces an efficient pipelined microarchitecture, including forwarding, interlocking and speculative execution. The generator may be extended, to produce further microarchitectural concepts, such as dynamic scheduling or sophisticated branch predictors. Such extensions would expand the microarchitectural design space. Additional processor implementations could be generated from the existing ViDL specifications.

[1] Development of an assembler generator, a disassembler generator and a linker generator for UPSLA; Generation and extension of an optimizing C compiler for the CoreVA VLIW processor.

Hardware sharing

There is a high optimization potential left to reduce the area requirement and power consumption of generated processors. The current generator implements a simple sharing method, which only merges functional units, if it does not have to introduce multiplexers. Basically, two functional units of the same kind can be merged, if they are not activated concurrently. A method for hardware sharing may be based on the existing method for port-assignment, which is similar in some respect.

Target languages

The existing generators produce a simulator in terms of C code and processors in terms of VHDL code. Further back-ends may be added, to produce for instance a processor specification in terms of Verilog code. Due to the modular design of generators, such extensions can be considered simple. As ViDL strictly abstracts from target languages, the existing specifications can be reused without any modification.

Web-based processor simulator

For the purpose of demonstration and education, a web-based processor simulator may be desirable, which runs in a web-browser. Technically, such a simulator may run on the client using JavaScript. To generate such a simulator, the simulator generator needs to be extended by a JavaScript back-end. The existing ViDL specifications are not affected by such an extension.

Parallel architectures

ViDL can be used to specify scalar processors, including vector and SIMD instructions. It may be extended in future, to support DSP instruction sets and VLIW instruction sets, such as the VLIW variant of CoreVA. An extension for VLIW may for instance define a set of execution slots and their relation to instructions. Scheduling constraints may be expressed using artificial resources. As an alternative, the set of valid schedules may be defined by a concrete grammar, where each non-terminal corresponds to one instruction.

Bibliography

[1] IEEE standard multivalue logic system for VHDL model interoperability (stdlogic1164). *IEEE Std 1164-1993*, 1993.

[2] ARM Limited. *ARM Architecture Reference Manual*, ARM DDI 0100E edition, 2000.

[3] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[4] J.L. Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 2009.

[5] C. Gordon Bell and Allen Newell. The PMS and ISP descriptive systems for computer structures. In *AFIPS '70 (Spring): Proceedings of the May 5-7, 1970, spring joint computer conference*, pages 351–374, New York, NY, USA, 1970. ACM.

[6] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, 1971.

[7] Anupam Chattopadhyay, Heinrich Meyr, and Rainer Leupers. Lisa. a uniform adl for embedded processor modeling, implementation, and software toolsuite generation. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, pages 95–132. Morgan Kaufmann, 2008.

[8] V.R. Dodani, N. Kumar, U. Nanda, and K. Mahapatra. Optimization of an application specific instruction set processor using application description language. In *Industrial and Information Systems (ICIIS), 2010 International Conference on*, pages 325 –328, 29 2010-aug. 1 2010.

[9] Charles Donnelly and Richard Stallman. *Bison - The Yacc-compatible Parser Generator*, 2010.

[10] Ralf Dreesen. ViDL specification of the Simple RISC (SRC) instruction set. `http://dreesen.net/vidl/specs/srisc/`, 2011.

[11] Ralf Dreesen, Michael Hußmann, Michael Thies, and Uwe Kastens. Register allocation for processors with dynamically reconfigurable register banks. In

Proceedings of the 5rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 5th IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2007), March 2007.

[12] Ralf Dreesen, Thorsten Jungeblut, Michael Thies, and Uwe Kastens. Dependence analysis of VLIW code for non-interlocked pipelines. In *Proceedings of the 8th Workshop on Optimizations for DSP and Embedded Systems (ODES-8)*, April 2010.

[13] Ralf Dreesen, Thorsten Jungeblut, Michael Thies, Mario Porrmann, Uwe Kastens, and Ulrich Rückert. A synchronization method for register traces of pipelined processors. In *Analysis, Architectures and Modelling of Embedded Systems*, volume 310 of *IFIP Advances in Information and Communication Technology*, pages 207–217. Springer Boston, 2009.

[14] Ralf Dreesen, Michael Thies, and Uwe Kastens. Type analysis on bitstring expressions. In *Proceedings of the 9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*, April 2011.

[15] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. pages 503 –507, mar. 1995.

[16] Jonathan D. Franz. An evaluation of CoWare Inc.'s processor designer tool suite for the design of embedded processors. Master's thesis, Department of Electrical and Computer Engineering, Baylor University, 2008.

[17] Markus Freericks. The nML machine description formalism. 1993.

[18] William F. Gilreath and Phillip A. Laplante. *Computer Architecture: A Minimalist Perspective*. 2003.

[19] R.E. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20(2):60 –70, mar/apr 2000.

[20] Gerhard Goos and William Waite. *Compiler Construction*. Springer, Jan 1984.

[21] Peter Grun, Ashok Halambi, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alexandru Nicolau. EXPRESSION: An ADL for system level design exploration. Technical Report 98-29, University of California, Irvine, 1998.

[22] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference, 1997. Proceedings of the 34th*, pages 299 –302, jun. 1997.

[23] Ashok Halambi, Peter Grun, Vijay Ganesh, and Asheesh Khare. Expression: A language for architecture exploration through compiler/simulator retargetability. In *In Proceedings of the European Conference on Design, Automation and Test*, pages 485–490, 1999.

[24] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2006.

[25] Vincent P. Heuring and Harry F. Jordan. *Computer Systems Design and Architecture (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[26] IBM. *Power ISA*, Version 2.04 edition, April 2007.

[27] CoWare Inc. *LISA Language Reference Manual*, 2008.

[28] MIPS Technologies Inc. *MIPS Architecture For Programmers Valume II-A: The MIPS32 Instruction Set*, 2010.

[29] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: an ASIP design environment. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 430 –436, 2000.

[30] Uwe Kastens. *Übersetzerbau*. Oldenbourg, 1990.

[31] Uwe Kastens, Peter Pfahler, and Matthias Jung. The eli system. In Kai Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 294–297. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0026439.

[32] Uwe Kastens, Anthony M. Sloane, and William M. Waite. *Generating software from specifications*. Jones and Bartlett Publishers, 2007.

[33] J. W. Klop, Marc Bezem, and R. C. De Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA, 2001.

[34] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[35] Kobayashi, Takeuchi, Kitajima, and Imai. Compiler generation in PEAS-III: an ASIP development system. In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems*, SCOPES 2001, 2001.

[36] Jae-Jin Lee, SeongMo Park, and NakWoong Eum. Design of application specific processor for h.264 inverse transform and quantization. In *SoC Design Conference, 2008. ISOCC '08. International*, volume 02, pages II–57 –II–60, nov. 2008.

[37] Steve Leibson. *Designing SOCs with Configured Cores*, 2006.

[38] U. Meyer-Baese, Guillermo Botella, Encarnacion Castillo, and Antonio Garcia. A balanced hw/sw teaching approach for embedded microprocessors. *International Journal of Engineering Education*, 26(3):584–592, 2010.

[39] Prabhat Mishra, Arun Kejariwal, and Nikil Dutt. Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models. *Rapid System Prototyping, IEEE International Workshop on*, 0:226, 2003.

[40] Rashid Muhammad, Ludovic Apvrille, and Renaud Pacalet. Evaluation of ASIPs design with LISATek. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5114 of *Lecture Notes in Computer Science*, pages 177–186. Springer Berlin / Heidelberg, 2008.

[41] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

[42] Vern Paxson, Will Estes, and John Millaway. *The flex Manual*, 2007.

[43] Jorgen Peddersen, Seng Lin Shee, Andhi Janapsatya, and Sri Parameswaran. Rapid embedded hardware/software system generation. In *In Proceedings of the International Conference on VLSI Design*, pages 111–116, 2005.

[44] Johan Van Praet, Dirk Lanneer, Werner Geurts, and Gert Goossens. nML: A structural processor modeling language for retargetable compilation and ASIP design. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, pages 65–93. Morgan Kaufmann, 2008.

[45] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Publishing Company, USA, 9th edition, 2009.

[46] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, March 1981.

[47] IEEE Computer Society. *IEEE Std 1364-2001 - IEEE Standard Verilog Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc, 2001.

[48] SPARC International, Inc. *The SPARC Architecture Manual*, SAV080SI9106 version 8 edition, 1992.

[49] Sun Microsystems, Inc. *UltraSPARC Architecture 2005*, Draft D0.9,15 edition, May 2007.

[50] Inc Tensilica. *Tensilica Instruction Extension (TIE) Language*, 2006.

[51] Henrik Theiling. Generating decision trees for decoding binaries. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 01*, pages 112–120. ACM, 2001.

[52] W.L. van der Poel. *The Logical Principles of Some Simple Computers*. PhD thesis, 56.

[53] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 184–188, New York, NY, USA, 2001. ACM.

[54] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004.

[55] O. Weiss, M. Gansen, and T.G. Noll. A flexible datapath generator for physical oriented design. In *Solid-State Circuits Conference, 2001. ESSCIRC 2001. Proceedings of the 27th European*, pages 393 – 396, 2001.

[56] Yoshiki Yamaguchi, Hung Tsoi, and Wayne Luk. FPGA-based smith-waterman algorithm: Analysis and novel design. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 6578 of *Lecture Notes in Computer Science*, pages 181–192. Springer Berlin / Heidelberg, 2011.

[57] Zilog, Inc. *Z8001/2 Z8000 CPU Product Specification*, 1985.