

The Vienna Architecture Description Language

FLORIAN FREITAG, LINUS HALDER, SIMON HIMMELBAUER, CHRISTOPH HOCHRAINER, BENEDIKT HUBER, BENJAMIN KASPER, NIKLAS MISCHKULNIG, and MICHAEL NESTLER, TU

Wien, Austria

PHILIPP PAULWEBER*, fiskaly GmbH, Austria

KEVIN PER, MATTHIAS RASCHHOFFER, ALEXANDER RIPAR, TOBIAS SCHWARZINGER, JOHANNES ZOTTELE, and ANDREAS KRALL, TU Wien, Austria

The Vienna Architecture Description Language (VADL) is a powerful processor description language (PDL) that enables the concise formal specification of processor architectures. By utilizing a single VADL processor specification, the VADL system exhibits the capability to automatically generate a range of artifacts necessary for rapid design space exploration. These include assemblers, compilers, linkers, functional instruction set simulators, cycle-accurate instruction set simulators, synthesizable specifications in a hardware description language, as well as test cases and documentation. One distinctive feature of VADL lies in its separation of the instruction set architecture (ISA) specification and the microarchitecture (MiA) specification. This segregation allows users the flexibility to combine various ISAs with different MiAs, providing a versatile approach to processor design. In contrast to existing PDLs, VADL's MiA specification operates at a higher level of abstraction, enhancing the clarity and simplicity of the design process. Notably, with a single ISA specification, VADL streamlines compiler generation and maintenance by eliminating the need for intricate compiler-specific knowledge. The original VADL implementation has a restricted copyright. Therefore, the open source implementation OpenVADL was started. This article introduces VADL, compares the original VADL implementation with the ongoing OpenVADL implementation, describes the generator techniques in detail and demonstrates the power of the language and the performance of the generators in an empirical evaluation. The evaluation shows the expressiveness and conciseness of VADL and the efficiency of the generated artifacts.

CCS Concepts: • **Software and its engineering** → **Architecture description languages; Retargetable compilers; Simulator / interpreter**; • **Hardware** → **Hardware description languages and compilation**.

Additional Key Words and Phrases: processor description language, compiler generator, assembler generator, simulator generator, hardware generator

ACM Reference Format:

Florian Freitag, Linus Halder, Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Benjamin Kasper, Niklas Mischkulnig, Michael Nestler, Philipp Paulweber, Kevin Per, Matthias Raschhofer, Alexander Ripar, Tobias Schwarzinger, Johannes Zottele,

*Research done at TU Wien

Authors' addresses: Florian Freitag, florian.freitag@student.tuwien.ac.at; Linus Halder, linus.halder@student.tuwien.ac.at; Simon Himmelbauer, simon@himmelbauer.net; Christoph Hochrainer, christoph.hochrainer@tuwien.ac.at; Benedikt Huber, benedikt.huber@tuwien.ac.at; Benjamin Kasper, benjamin.kasper@student.tuwien.ac.at; Niklas Mischkulnig, niklas.mischkulnig@student.tuwien.ac.at; Michael Nestler, michael.nestler@yahoo.com, TU Wien, Vienna, Austria; Philipp Paulweber, ppaulweber@fiskaly.com, fiskaly GmbH, Vienna, Austria; Kevin Per, kevin.per@student.tuwien.ac.at; Matthias Raschhofer, matthias.raschhofer@student.tuwien.ac.at; Alexander Ripar, alexander.ripar@student.tuwien.ac.at; Tobias Schwarzinger, tobias.schwarzinger@tuwien.ac.at; Johannes Zottele, johannes.zottele@tuwien.ac.at; Andreas Krall, andi@complang.tuwien.ac.at, TU Wien, Vienna, Austria.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).

Manuscript submitted to ACM

Manuscript submitted to ACM

and Andreas Krall. 2024. The Vienna Architecture Description Language. 1, 1 (February 2024), 81 pages. <https://doi.org/XXXXXXX.XXXXXX>

1 INTRODUCTION

The Vienna Architecture Description Language (VADL) is a Processor Description Language (PDL). The name is inspired by the Vienna Definition Language (VDL) which was developed 50 years ago for the formal specification of the programming language PL/I using operational semantics [122].

Why do we need another PDL? The development of backends for compilers like LLVM, GCC or just-in-time (JIT) compilers is cumbersome and error prone. The specifications are huge and very difficult to understand even for experienced compiler developers. Thus, we want to get all these distinct compilers automatically generated based on a single concise specification. Additionally, we also want to automatically produce an assembler, a debugger, disassembler, Instruction Set Simulator (ISS) and linker. To the best of our knowledge, currently there does not exist any PDL or compiler backend specification language that achieves this. Furthermore, we want to do computer architecture research and teaching on a higher level of abstraction compared to what, we believe, other existing PDLs or Hardware Description Languages (HDLs) currently offer. The low level micro architecture specification means provided by current PDLs or HDLs lead to huge unmaintainable specifications where the instruction's semantics and the microarchitecture are intermingled. It should be possible that hardware, Cycle Accurate Simulators (CASs) and instruction schedulers for compilers are also automatically generated from such a high level specification. We want to push forward the research in the area of PDLs. For all these reasons we designed VADL and developed the necessary generator technologies.

VADL permits the complete formal specification of a processor architecture. Additionally it is possible to specify the behavior of generators which produce different artifacts from a processor specification. From a single concise VADL processor specification, the VADL system is able to automatically generate an assembler, a compiler, linker, functional ISS, CAS, synthesizable specification in a HDL, test cases and documentation. VADL strictly separates the Instruction Set Architecture (ISA) specification from the Microarchitecture (MiA) specification. The ISA specification is needed by all generators. The MiA specification is used by the HDL and CAS generators as well as for instruction scheduling in the compiler. An ISA specification can be implemented by one or more MiA specifications. The Application Binary Interface (ABI) specification defines a programming model and is used by the compiler generator.

VADL has been designed to enable concise comprehensible specifications. A novice should be able to understand a specification without prior knowledge of VADL. Redundant specifications are avoided. VADL is a safe language. It is strongly statically typed. The language parser and the artifact generators apply extensive consistency checks. VADL is a generator language, executable specifications are not possible.

1.1 Contribution

The development of VADL led to innovations across different domains. Our main contributions are:

- A concise and comprehensible processor description language
- A high abstraction level ISA independent MiA specification language
- ISA to MiA mapping by using inherent properties of the instruction's behavior for MiA assignment
- A simple pred-LL(k) parsable *syntactical pattern-based* macro system
- Syntactic type safe higher-order macro templates
- Specification of assembly language by string expressions

- Assembler generation by automatic grammar inference through program inversion
- Compiler generation by automatic pattern inference from operational semantics specifications
- MiA synthesis by reduction of the instruction's data flow graph
- MiA hazard detection and optimization

See section 2 for a short explanation of the concepts listed here. Additionally, we present a variety of smaller contributions particularly useful for our exploratory language design of VADL:

- Composable syntax types using *records* and type aliases
- Constraints on instruction encodings
- Register file aliases with different constraints
- Access functions for decoding and encoding of format fields
- Concise specification of user mode emulation

Additionally, we have started to develop *OpenVADL* which is the open source implementation of VADL.

1.2 Outline

Section 2 gives some background information about the different domains touched upon in this article. This section can be skipped by readers who already have a deep knowledge of compilers and computer architecture. Section 3 presents the most important language elements of VADL by examples. Section 4 describes the implementation of the VADL compiler, its intermediate representation and the different generators in detail. Section 5 does a detailed qualitative and quantitative evaluation of VADL and its generators. Section 6 compares VADL and its implementation with related work.

2 BACKGROUND

Design Space Exploration (DSE)

One of the most relevant applications for a PDL is DSE for Application Specific Instruction Set Processors (ASIPs). If it is possible to *automatically* generate a set of tools from an architecture description in a PDL, the productivity of the architecture design process can be improved by establishing short feedback loops between design iterations. The set of these tools should at least contain a compiler toolchain, a CAS and a synthesizable hardware model in a HDL. Additionally it can also include a functional ISS, test cases, documentation and other artifacts. As described in [92] this establishes two feedback loops:

- The compiler toolchain together with the CAS can provide accurate performance information for a given software workload.
- The hardware model can provide information about the maximum clock frequency or chip area needed for the actual hardware implementation.

To consider this information in the architecture design is especially useful in embedded systems and ASIPs design, because of stringent hardware constraints and the known workload.

Compiler Toolchain

VADL can generate the toolchain necessary for producing machine executable code from a high level programming language like the C programming language. The main tools in this toolchain are the compiler, the assembler, and the linker.

Compiler. In general, a compiler translates a program written in one programming language into a semantically equivalent program in another programming language. In the most common case, and the case relevant for VADL, the compiler translates a program written in a high-level and architecture *independent* programming language into an architecture *dependent* assembly language. An assembly language is a direct textual representation of machine code. In other words, there is a direct correspondence between machine instructions and assembler instructions.

Typically, high-level programming languages offer syntax constructs to describe control flow and data structures in a manner easily understood by a programmer. This abstraction helps to facilitate reasoning about the semantics of an implemented algorithm and hides unnecessary details. It also helps in the maintenance of software. An assembly language typically does not provide such syntax features and is more difficult to read and maintain for a programmer. The primary purpose of the compiler is to translate from the higher abstraction level of the programming language to the lower abstraction of the assembly language. This translation is typically done in multiple passes. The compiler can employ various optimization steps during translation to increase the resulting program's performance or reduce its code size.

A *retargetable* compiler is designed to make it easy to add support for a new target architecture. The design of a retargetable compiler has target architecture *independent* and target architecture *dependent* components. Target architecture independent components implement common transformations and optimizations for all targets, e.g., dead code elimination. Target-dependent transformations, like register allocation, are implemented such that the main algorithm is target-independent but can be parameterized with target-specific data.

A retargetable compiler is often roughly subdivided into three parts.

- The *frontend* is input language dependent. It typically parses and analyzes a high-level programming language as input. It abstracts each concrete programming language to a common Intermediate Representation (IR). This IR is input language independent and it is used in the later stages of the compilation process. To support multiple high-level programming languages, the front end has to be able to abstract all of them to the common IR.
- The *middleend* operates on the IR. It performs transformations and optimizations that are common to all target architectures. It is independent of both the input programming language and the target architecture. In order to facilitate retargetability, as much functionality as possible should reside in the middleend.
- The *backend* emits code specific to the target architecture. It has specialized functionality for each supported target.

Parser. An important component of the frontend in a compiler is the parser. In the context of VADL an important class of parsers are *predicated LL(*) parsers*. These parsers read input from *left-to-right* and derive the *leftmost* non-terminal symbol, hence *LL*. The lookahead is arbitrary but finite – hence *** – as opposed to *LL(k)*, where the lookahead is bounded by *k*. A special case of *LL(k)* is *LL(1)*, which is the most resource-efficient.

A *predicated* parser can also use syntactic predicates to decide which derivation to apply. This allows the parser to resolve possible divergences, and it can help to make the input grammar more readable and maintainable.

Also part of the frontend is the *macro system*. Macros are a well established way to realize language extensions. In general, a macro is a user defined procedure that reads and transforms program code. In its simplest form, a macro can define simple string replacements. A complex macro on the other hand can be seen as a form of generative programming. Since a macro is also defined as program code, it can also be processed by a macro. This leads to the idea of *higher-order* macros. So, a macro is higher-order if it takes another macro as input, or returns a macro or both.

Lexical macros are string operations that operate directly on the text of the program code. As such, they are not aware of program structure. VADL macros are *syntactic macros* that are aware of the syntactic structure and operate on the Abstract Syntax Tree (AST). In addition, VADL macros support a type system for syntax types, where each syntax element is of a certain type. The type system helps prevent programming errors and enhances readability. The types in VADL macros are extensible with programmer-defined *composable syntax types*, also known as *records*. A record is a type checked collection of named syntax elements. A macro can access a member of a record via its user-defined name.

For more information about the VADL parser and macro system see section 4.2.

Instruction Selection. An important aspect of the compiler backend is the translation from the target independent IR to the target specific assembly language. This is done by *selecting* the instructions offered by the target in such a way that the resulting program is functionally equivalent to the IR input program. Hence the name of the pass *instruction selection*. The most common approach to achieve this is described in the standard compiler text book [7] and also implemented in LLVM which is used in VADL's compiler generator. The idea is to represent the IR input program as dependency graph – the instruction selection graph – where each node is an IR instruction and each directed edge represents a dependency. Each target instruction is represented as a graph, which represents the input, the output and the semantics: the instruction pattern. A pattern is a potential subgraph of the instruction selection graph. The instruction selection pass has to find a complete cover of the instruction selection graph, by tiling the appropriate patterns. In a VADL ISA specification, the semantics of an instruction is formulated as operational semantics. Thus an important task of VADL's LLVM Compiler Backend (LCB) is to infer the instruction patterns to be used during instruction selection from the VADL specification. See section 4.5 on implementation details.

Register Allocation. The compiler has to decide which values to keep in registers for fast access, and which values to store in memory, resulting in slower access. This is done in the register allocation pass. The *live range* of a value in a program is the collection of all points in the program between the definition of a value and its last use. If the number of overlapping live ranges is greater than the number of available registers, the compiler has to break up live ranges and insert *spill code* – i.e., store a value in memory and load it when it is needed again.

A common approach for register allocation is *graph coloring*, described in [30]. The idea is to construct an interference graph, representing live ranges as nodes and interference relations as edges. Each available register is represented by a color. The goal of the approach is to assign each node a color, such that no two nodes with the same color are connected by an edge.

OpenVADL's QEMU Generator uses the same approach to minimize the number of temporary variables, see section 4.8.7.

Instruction Scheduling. While respecting all dependencies and keeping the original semantics of the program, the order in which the instructions are arranged can be subject to optimization. This pass is called *instruction scheduling*. Skillfully rearranging the order of the instructions can lead to better resource usage, and fewer pipeline stalls, thus improving performance. Especially in target architectures with explicit instruction level parallelism, like Very Long Instruction

Word (VLIW) architectures, a good instruction schedule is crucial, as the instruction scheduler has to decide which instructions should be executed in parallel.

A Basic Block (BB) is a sequence of instructions with one entry point, and one exit point, and no other control flow. In most cases, the instruction scheduler operates on a single BB. This limits the scope of the scheduler and restricts its possibilities. One technique to improve this situation is to merge BBs with *control flow elimination*, thus increasing the scope of the scheduler. This is done by transforming control dependencies into data dependencies.

This can be done by using predicated instructions, which have to be available in the target architecture. Alternatively it can be realized by speculative execution and conditional assignment. However, this only works if the speculated instructions do not have side effects.

VADL's MiA synthesis also uses control flow elimination, see 4.9. Again, the idea is to replace control dependencies with data dependencies.

Compiler Optimizations. Optimizations done by the compiler are program transformations that retain the semantics of a program, while improving it according to some metric. Typical metrics of interest are: Execution time, memory use, code size or power intake. Often trade-offs have to be made, because a given transformation may improve one metric, but harm another.

In order to facilitate code reuse, it is good engineering practice to place program optimizations in the middleend of the compiler, if possible. Due to their target independent nature, optimizations in the middleend can benefit all supported target architectures. Often optimizations in the middleend are not entirely target independent, but are parameterized with information about the target architecture. These can help the – otherwise target independent – middleend pass make some optimization decisions.

One important aspect regarding the middleend is the design of the IR. The IR should make it easy to perform program analysis and program transformations. An IR in Static Single Assignment (SSA) form can help with these goals. In SSA form, a variable is assigned a value *exactly once* and cannot be changed thereafter. This property greatly facilitates dependency analysis, and determining *use-def chains*, since there is a single point of definition for a variable. Many widely used compilers employ an IR in SSA form, e.g. GCC [116] and LLVM [77], which VADL uses. The article [35] describes how to efficiently generate the SSA form of a given program, and the topic – including many optimizations – is covered comprehensively in the SSA book [106].

An optimization well suited for SSA form is *global value numbering* [28]. The goal of global value numbering is to remove redundant computations that compute the same value, by replacing them with the value itself. It does so by labelling expressions that produce the same value with the same *value number*. This facilitates analyzing inputs and outputs of computations, thus making it possible to determine which computations yield the same value.

Common Subexpression Elimination (CSE) has a similar goal, as it also tries to eliminate redundant computations. It does this by analyzing which expressions are available at which point in the program. Then it decides where to replace identical – i.e., common – expressions by a single variable holding the computed value of the expression [32].

Important and common compiler optimizations are those operating on constant values. A constant value in a program does not depend on the program's input. Thus, everything needed to compute it is already known at compile time; hence the name *compile time constant*. Programs often contain expression that only depend on compile time constants. Therefore, it is possible to compute their value at *compile time*, thus moving complexity from the program's runtime to its compile time. This optimization is called constant expression evaluation or *constant folding*.

Working closely together with constant folding is *constant propagation*, which replaces the use of a variable, which the compiler can determine to be constant, with its constant value. This in turn may lead to new opportunities for constant folding. Both of these techniques and many more are described in the book [94].

Dead code is program code that is guaranteed to never be executed. *Dead code elimination* is a compiler optimization that tries to find dead code, and remove it. One common application arises when control flow depends on a compile time constant condition and the compiler can guarantee that a branch target can never be reached.

Sometimes it is beneficial to move computations in a program to another point within the program. This is called *code motion* and may serve to provide the result of a computation to more places where this result is needed, without recomputing it. Alternatively it can be employed to prevent recomputations inside a loop where the result does not change between loop iteration, known as *loop invariant code motion*. Code motion has to be done carefully, as it can lengthen live ranges, or increase code size. Thus, sophisticated approaches like *lazy code motion* [74] have been developed to make sensible code motion decisions.

In general function calls cause an overhead, as the program has to prepare the stack frame and handle input parameters and return values. *Function inlining* is a way to remove this overhead. The idea is to replace the function call with an inlined copy of the callee. Thus the complete functionality of the callee is available in the caller without the overhead of a function call. This can also increase the scope on which the instruction scheduler may operate. However, function inlining typically leads to an increase in code size.

Some compiler optimizations are target machine *dependent* and are considered to be part of the backend. In many target architectures it is the case that some operations need more resources than others. *Strength reduction* tries to replace expensive operations with cheap operations which yield the same result. A typical example is integer multiplication by a power of two. If the value is represented as two's complement, the compiler can choose to use a bitwise left-shift instead, which is typically cheaper than multiplication. Strength reduction is presented in [33]. This optimization is an example of a pass that can reside directly in the target dependent backend, or, with appropriate parameterization, in the middleend.

Assembler. The assembly language is independent of the concrete binary encoding of the machine instructions. The assembler reads a textual representation of the machine program, i.e., a program in assembly language, and generates a binary representation of this program that a processor can execute. This step consists of encoding the machine instructions as a bit pattern. The main task of the assembler is to apply such a binary encoding to the assembly program, thus creating an object file.

Since the assembler has to parse a string representation of the program, a straight forward way is to generate a parser from a grammar describing the assembly language. However, sometimes it is not the most convenient way to do this. An ISA specification in VADL already contains a specification of how to print each instruction as a string in the assembly language. This functionality is also called a *pretty printer*, as it takes a binary encoded instruction and prints it in readable form. If the assembly language is not too complex, it is possible to deduce the grammar of the assembly language from the specification of the pretty printer. Informally speaking, it is possible to go *the other direction*, and generate a parser from the specification of a pretty printer. This technique is called *program inversion*. This feature helps to keep VADL specifications succinct and reduce redundancies. For more complex assembly languages, VADL also supports explicit specification of a grammar for the assembly language. See section 4.6 for details.

Linker. The linker joins object files together, creating a single *executable* native program that can be run on a processor. The linker has to resolve symbolic addresses and assign them concrete address values from the machine's address space.

It must also place the object files containing executable code in non-overlapping memory segments so each instruction has a unique address. This process is called *relocation*. Often, target-specific rules for relocations exist that have to be obeyed by the linker.

Microprocessor

In the context of VADL, a microprocessor is an integrated digital electronic circuit that reads data from memory, executes operations on these data and writes data to memory. These operations are called *instructions*. The representation of these instructions, together with initial values in memory, is called a *native program*. Which instructions are available and how they are represented in memory is defined by the ISA. Thus, a microprocessor *implements* an ISA. The MiA describes how the implementation for a microprocessor is realized, specifically, it defines what components make up its internal structure and how they interact.

The timing behavior of these designs is based on clock signals. While modern processors do use multiple clocks with different frequencies, VADL currently focuses on designs of processors with a single clock domain. Related to the clock cycle, the machine cycle is the time interval between the start of two instructions [113, p. 51]. Depending on the MiA this can be longer than one clock cycle, e.g., for multi-cycle MiAs.

Besides low-level hardware design decisions the MiA is fundamental to reach certain design goals, such as performance or power consumption, in the hardware implementation of a microprocessor. Techniques used for this include pipelining, forwarding and branch prediction, but also advanced techniques specific to superscalar and out-of-order processors, like register renaming, reservation stations and reorder buffers.

Pipelining is concerned with splitting instruction execution into smaller steps structured in stages, which allows resulting hardware to run faster, i.e., at higher clock frequencies. In a classical scalar pipeline this results in different steps of multiple instructions being executed at each point in time and gives rise to data hazards, that occur when instructions require results from previous instructions still in execution. To resolve this problem without delaying instructions a MiA can be equipped with *forwarding logic*, which connects stages that read values with other stages where the needed results are already available.

Another problem that is caused by the simultaneous execution of sequential instructions occurs in the presence of branch instructions. In order to still execute further instructions while the result of a branch is not yet known, processors employ *branch predictors*. These predictors speculate on the outcome of the branch – i.e., the branch predictor tries to anticipate the correct target address of the branch. The processor continues execution of subsequent instructions at the predicted target address. Mispredicted branches must be resolved by either not changing the state during speculative execution or by rolling back changes.

Superscalar processors strive to complete more than one instruction per cycle. To achieve this they implement out-of-order execution. The processor here analyzes the data dependencies between the fetched instructions and executes the instructions it can simultaneously. At the core of such architectures are multiple functional units that handle the computations involved in the instructions in parallel.

For the processor to be able to continue decoding instructions while others wait for their input operands to become available from previous computations, it uses *reservation stations*. These are buffers that collect the waiting instructions and their operand values and then move instructions to functional units when their operands are available.

The *reorder buffer* keeps track of the unfinished dispatched instructions. This buffer is used to complete instructions in order and is also used when doing speculative execution.

To exploit instruction-level parallelism even more, superscalar architectures try to reduce anti- and output dependencies by *renaming registers*. The design then contains more physical than architectural registers. If a new instruction has an anti- or output dependence on an instruction that has not completed yet, its output register is renamed and the instruction can be dispatched right away. This renaming can be realized using a separate register rename file or be integrated with the reorder buffer, recording a rename register for every instruction in progress.

VADL can be used to describe many of these micro-architectural aspects in its *MiA* section. To implement a MiA, conventionally an HDL is used to represent the design at Register-Transfer Level (RTL).

Register-Transfer Level. This is the level of abstraction where the design is described as a set of registers and their connection through combinational logic. The registers are the only components having memory, i.e., are solely holding state. Their inputs are consequently outputs from other registers and external inputs wired through logic gates. Typically the description also involves defining a clock signal and reset logic (initial values) for the registers. In contrast to mere combinational logic, such a design with memory embodies sequential logic – its outputs are not only determined by its current inputs, but also the sequence of past inputs.

Since the internal state and its possible transitions in sequential logic is not always obvious on RTL, a useful abstraction for creating sequential logic is the definition in the form of a state machine (following the formal model of finite-state machines). State machines are defined by a set of states with corresponding output and next state logic. Such a state machine can then be implemented using registers to hold the state and combinational logic to feed outputs and the next state logic.

Verilog and Very High Speed Integrated Circuit Hardware Description Language (VHDL) are examples of widely used low-level HDLs. These languages can specify a wide range of circuits and also behavior not implementable in hardware (non-synthesizable behavior). Hence, they are not only capable of describing microprocessors.

VADL emits *Chisel* [15], which then generates Verilog. Chisel’s goal is to offer convenient abstractions and to be easier to work with than directly writing Verilog code. For instance, it only generates synthesizable Verilog.

Chisel. Chisel is an HDL used to describe digital circuits on the RTL. It tries to increase the circuit designer’s productivity by providing powerful abstractions compared to the low level languages like Verilog or VHDL. Many of these abstractions are a result of the fact that Chisel is a Domain Specific Language (DSL) embedded in the general purpose programming language *Scala*. Thanks to this, Chisel can directly use Scala’s object oriented and functional programming features as well as parameterized types and type inference. Chisel’s goal is to use these features to make the hardware design shorter and easier to maintain and extend. The Chisel compiler emits Verilog code which in turn can be used to map to Field Programmable Gate Arrays (FPGAs) or for Application Specific Integrated Circuit (ASIC) synthesis.

Simulation

ISA simulation is the process of executing a program on a software implementation of the target ISA instead of a hardware implementation, i.e., a processor. This software implementation is called a *Simulator*. For the properties and behavior under consideration, the behavior and execution of the simulator is identical to the simulated processor. However, certain aspects might not be simulated depending on the requirements of the simulation. For instance, the goal of an ISS is to match the semantics of the ISA but without considering the behavior of an underlying microarchitecture. This is sufficient for executing programs written for the target ISA, but e.g. analyzing energy consumption of the processor will not be possible. Whether a certain property or behavior should be modeled by the simulator is an important design decision.

Hence, VADL aims to describe various architectures and generate various simulators depending on the user’s needs. The most relevant approaches for VADL are described in the following paragraph.

The program that is executed by the simulator is called the *guest*, the system running the simulator is called the *host*. A simulator that primarily takes care of the *semantics* of each simulated instruction is called an Instruction Set Simulator (ISS). In addition, a simulator can also model certain other aspects that may be of interest, particularly the performance metrics of the simulated processor. A simulator that can also take the MiA of a particular processor into account and simulate the complete processor pipeline is called a Cycle Accurate Simulator (CAS). A CAS has to handle forwarding behavior, pipeline stalls, cache/memory latencies and other MiA related aspects of a processor. This is why a CAS is usually more complex than an ISS and its execution is computationally more costly.

There are several ways to implement a simulator. The most straightforward approach is *interpretive simulation*. Here, the simulator reads the guest program instruction by instruction and simulates the effect of each instruction. Since all simulation decisions happen at the time of simulation, it is a very flexible approach, but also computationally costly. VADL’s ISS generator, presented in section 4.7, emits a simulator that falls into this category.

Another approach that tries to achieve better simulation performance is compiled simulation [91]. It translates the guest program into a program that is executable directly on the host, while keeping the same functional behavior. This way it moves complexity to compile time and also makes it possible to apply optimizations during compilation. Normally compiled simulation does not allow self-modifying code in the guest program, as it is compiled ahead of time.

Dynamic Binary Translation (DBT) [31] tries to combine the flexibility of interpretation with the performance of compiled simulation. It only translates frequently executed code fragments into executable host code. In this regard it is conceptually similar to a JIT compiler. Its dynamic nature also allows it to handle self-modifying code. QEMU [19] uses DBT, thus the simulator emitted by OpenVADL’s ISS generator, presented in section 4.8, falls into this category.

Both, processor specifications and simulators have to be validated. The most common technique is *co-simulation* where the execution state of a test application on one simulator is compared against the execution state on another simulator or on real hardware [52]. The comparison of the execution state can be done immediately after the execution of every instruction. An alternative is to generate execution traces and to compare the traces. The validation can be done at different levels of accuracy depending on the kind of execution state to be checked. The state just can be the content of the modified registers or can contain internal processor states like pipeline registers and microarchitecture elements.

3 THE PROCESSOR DESCRIPTION LANGUAGE

3.1 Introduction

The purpose of the PDL VADL is the complete specification of a processor architecture regarding the instruction set, the microarchitecture, the application binary interface, the assembler, the compiler, the linker, a functional ISS, a CAS, a synthesizable specification in a HDL, test cases and documentation. The aim of VADL is to facilitate the development and customization of processors and their corresponding toolchains. Thus, VADL enables rapid DSE of ASIPs, leading to higher quality processors and tools at reduced development costs and shorter time to market. We want to highlight that even for existing architectures VADL can be used solely for generating compilers for systems like LLVM or GCC, avoiding the need for LLVM or GCC specific know-how.

VADL is a PDL, that is a DSL in the domain of computer architecture and compiler construction. Potential users are computer architects or compiler developers with an academic or industrial background, but they do not require

extensive knowledge in both fields. Nonetheless, all use cases should be served by a single language. The language must provide an easily comprehensible syntax and semantics. A user without prior knowledge of VADL should understand a specification of a moderately complex architecture at first sight. Therefore, the behavioral parts of VADL are inspired by Java, C++, Rust and Chisel enriched with ideas from functional programming to provide familiarity to the users. VADL has a unique syntax and static semantics.

There is a long-standing debate whether a specification language should be executable [50, 61]. Executable specifications only work well if the language is single purpose, e.g. used for the specification of simulators. A VADL behavior specification has to fulfill multiple purposes with the same specification, describing the semantics of a compiler's code generator, the semantics of simulators and the instruction execution in hardware. Therefore, a VADL processor specification cannot be executed directly, but executable artifacts are produced by generators and thus VADL is a generator language.

VADL is a specification language where a processor can be described on a high abstraction level. The goal is to have a concise specification that is easy to write by the user and, at the same time, easy to analyze by the generators. The implementation of a concrete generator should not have any influence on the design of VADL.

3.1.1 Strict Separation of ISA and MiA. VADL strictly separates the specification of the ISA and the specification of a concrete MiA implementation. Different implementations can exist for the same ISA specification, realized by different MiA specifications. This strict separation follows the best practice design process in computer architecture introduced by Fred Brooks with the architecture of the IBM System/360 [8] and advocated by Richard Sites, the chief architect of the Alpha AXP architecture [115]. The ISA part specifies the register sets and the behavior, encoding and assembly language representation of the instructions, while the MiA part describes the structure and microarchitecture of the processor. Regarding the commonly used classification of PDLs into structural, behavioral, or mixed languages, the ISA is related to the behavioral part, and the MiA is related to the structural part and therefore, VADL is a mixed PDL. The MiA description of VADL operates on a higher level of abstraction than existing structural PDLs.

There are no references from the ISA part to the MiA part. Only some references from the MiA part to the ISA part are allowed. The ISA part is sufficient to generate a purely functional ISS or, together with an ABI, a compiler. The MiA part is necessary to synthesize hardware, to generate a CAS or to generate an instruction scheduler for a compiler.

3.1.2 Language Safety. VADL is designed with high productivity and type-safety in mind. Therefore, the language is strongly statically typed, but type inference is supported to keep the specification concise. In addition, static analysis prevents VADL developers from writing illegal specifications. For example, format fields are not allowed to overlap and a register write cannot occur before a read in the semantics of an instruction. Furthermore, VADL supports syntactic macros which are also type checked.

3.2 Overview

VADL provides a Chisel-like type system to represent arbitrary bit vectors. There are two primitive data types – `Bool` and `Bits<N>`. `Bool` represents Boolean typed data. `Bits<N>` represents an arbitrary bit vector data type of length `N`. Furthermore, to explicitly express signed and unsigned arithmetic operations VADL provides two sub-types of `Bits<N>` – `SInt<N>` and `UInt<N>`. `SInt<N>` represents a signed two's complement integer type of length `N`. Note that the length of this signed integer data type contains the sign-bit and data bits. `UInt<N>` represents an unsigned integer type of length `N`. For all bit vector based types – `Bits`, `SInt`, and `UInt` – VADL will try to infer the bit size from the surrounding usage. But for definitions, a concrete bit size has to be specified in order to determine the actual size of, e.g., a register.

In contrast to Chisel the size of the resulting bit vector of an operation is identical to the size of the source operands. An exception is the multiplication where two versions are available, one with a result with the same size and one with a double sized result. An additional String type is available in the assembly specification and the macro system.

```

1  constant MLen = 32
2
3  using BitsM = Bits<MLen>
4  using SIntM = SInt<MLen>
5  using UIntM = UInt<MLen>
6
7  function lessthan (a: SIntM, b: SIntM) -> Bool = a < b
8
9  import rv32i :: RV32I
10
11 instruction set architecture RV32IM extending RV32I = {}
12
13 application binary interface ABI for RV32IM = {}
14
15 assembly description Assemble for ABI = {}
16
17 micro architecture FiveStage implements RV32IM = {}
18
19 micro processor CPU implements FiveStage with ABI = {}
20
21 user mode emulation UME for CPU = {}

```

Listing 1. VADL specification

Listing 1 shows the main elements of a VADL processor specification. Usually, a VADL processor specification has some global definitions in the beginning, followed by some sections describing ISA or MiA, which are described in more detail in the following sections. On line 1, a constant `MLen` with the value 32 is defined. Type aliases can be defined with the keyword `using` as shown on lines 3 to 5. On line 7, a function is defined that compares two values of type `SIntM` and returns the result of the comparison as a value of type `Bool`. `import` allows the import of VADL specification parts from separate files. On line 9, a specification named `RV32I` is imported from a file called `rv32i.vadl`. In this example, `RV32I` refers to another ISA specification.

An `instruction set architecture` specification can extend another ISA specification (line 11). Section 3.3 contains a detailed description of the ISA specification. Lines 13 to 21 demonstrate the definition of the `application binary interface` (see Section 3.5), the `assembly description` (see Section 3.7), the `micro processor specification` (see Section 3.8) and the `user mode emulation` (see Section 3.6). On line 17 a MiA named `FiveStage` implements `RV32IM` (see Section 3.4).

More complex examples and the VADL grammar are available at the [VADL home page](#).

3.3 Instruction Set Architecture Section

The ISA section is the major part of a processor specification. Listing 2 gives a small example specifying a subset of the RISC-V architecture with one branch instruction. The section starts with the keyword `instruction set architecture` followed by the name of the architecture, which we simply call `RV32I`. On line number 3, a constant `Size` with the value 32 is defined. Constant expressions can be used in a constant definition, as demonstrated on line number 4. These constant expressions are evaluated during parsing.

Type casting is done with the keyword `as`, shown on lines 28 and 30. Type casting does zero extension, sign extension, or truncation of values if necessary.

Memory is defined on lines 12 to 14 by the mapping of a 32-bit address to an 8-bit byte. The memory definition shows the use of annotations in square brackets. Annotations can be applied to most of the definitions. Annotations for memory are `[littleEndian]` and `[bigEndian]` which are allowed to be used in a dynamically evaluated expression, e.g., depending on the value of a configuration register. A further memory annotation is the memory consistency model, e.g., `[sequentialConsistency]`, `[totalStoreOrdering]`, or `[rvWeakMemoryOrdering]`. There exist only a small number of predefined consistency models. Predefined microarchitectural elements like cache protocols know which consistency models they obey and the correct combinations are verified by the type checker. If more than one memory type is defined one has to be declared as instruction memory. It is also verified the type of the program counter fits to the address type of the instruction memory.

Declaring a Program Counter (PC) (line 17) is mandatory. In most architectures, the PC points to the start of the current instruction when used inside an instruction specification (lines 16 to 17). This behavior can be changed by adding the annotation `[next]`, which lets the PC point to the end of the current instruction. The ARM AArch32 architecture has the peculiar behavior that the PC points to the end of the following instruction, which can be specified by the annotation `[next next]`. If an instruction does not explicitly modify the PC, it is implicitly incremented by the instruction size in each execution cycle.

The RISC-V RV32I architecture has an integer register file named `X` with 32 registers, which are 32 bits wide (lines 19 to 20). The register with index 0 is hardwired to the value 0. This can be specified with an annotation that maps the constant 0 to the specified register.

Instruction words or system registers are commonly split into multiple fields. The VADL format definition allows the specification of such instruction or register formats with their corresponding fields. They can either be specified by connecting names with bit positions (as in Listing 2 lines 23 to 27) or by connecting names with types (as in Listing 3 lines 2 to 6). Sometimes fields in instruction words are not used directly, e.g., an immediate value is sign-extended or a register index can access only the higher half of a register file. For convenient use of such fields, access functions can be defined. In Listing 2 line 28, the access function `immS` is defined, which sign-extends the field `imm` to `Size1` (31) bits and concatenates it with a binary constant of type `Bits<1>` and value 0. The binary comma operator applied in parentheses defines a bit vector (line 28) concatenation or a string (line 45) concatenation. During instruction selection, a compiler must know what immediate values are valid and how they can be encoded. For trivial access functions, the compiler generator can determine the validity and encoding function of values. For nontrivial access functions, the validation and encoding functions have to be specified in the predicate and encoding part of the format specification. The validation on line 30 specifies that the value must be a multiple of 2 and must be in the range from -4096 to 4095. The encoding on line 33 specifies that `imm` is a bit slice of `immS` from position 12 to position 1.

With the instruction definition, the behavior of an instruction is specified (lines 37 to 42). Every instruction has a name (BEQ in the example) and an instruction format type (Btype in the example). Between the curly braces (lines 39 to 41), statements in a style inspired by functional programming languages specify the behavior. Assignments denoted by the assignment operator `:=` are possible only to registers and memory locations. To each variable, only one assignment is allowed, i.e., the behavior is *single assignment*. All reads to a register must occur before any writes to this register. The same is true for a given memory location. This requirement is checked by the VADL language parser. The `let` statement defines a constant (`cond` in the example) that can be used in the (block) statement after the keyword `in`. The else part of the *if statement* is optional but required for an *if expression*. Multiple conditional expressions can be

```

1  instruction set architecture RV32I = {
2
3      constant Size = 32 // architecture size is 32 bits
4      constant Size1 = Size - 1 // architecture size minus 1
5
6      using Byte = Bits< 8 > // 8 bit Byte
7      using Inst = Bits< 32 > // instruction word type
8      using Regs = Bits<Size> // register word type
9      using Addr = Regs // address word type is equal to the register type
10     using Index = Bits< 5 > // 5 bit register index type for 32 registers
11
12     [littleEndian] // memory is accessed little endian
13     [rvWeakMemoryOrdering] // RISC V weak memory ordering
14     memory MEM : Addr -> Byte // byte addressed memory
15
16     [current]
17     program counter PC : Addr // PC points to the start of the current instruction
18
19     [X(0) = 0] // register with index 0 always is 0
20     register file X : Index -> Regs // integer register file with 32 registers
21
22     format Btype : Inst = // Btype instructions are Inst sized
23     { imm [31, 7, 30..25, 11..8] // 12 bit immediate value
24       , rs2 [24..20] // 2nd source register index
25       , rs1 [19..15] // 1st source register index
26       , funct3 [14..12] // 3 bit function code
27       , opcode [6..0] // 7 bit operation code
28       , immS = (imm as SInt<Size1>, 0b0) // shifted and sign extended immediate value immS
29       : predicate { // immS is a multiple of 2 and -4096 <= immS <= 4095
30         immS => (immS(0) = 0) & (((immS as UInt) + 4096) <= 8191)
31       }
32       : encode {
33         imm => immS(12..1) // slice bits 12 to 1 from immS to encode imm
34       }
35     }
36
37     [operation BranchOp] // BEQ belongs to the set of branch operations
38     instruction BEQ : Btype = { // branch equal instruction
39       let cond = X(rs1) = X(rs2) in
40       if cond then
41         PC := PC + immS
42       }
43     [rs1 != rs2] // source register indices should be distinct
44     encoding BEQ = {opcode = 0b110'0011, funct3 = 0b000}
45     assembly BEQ = (mnemonic, " ", register(rs1), ", ", register(rs2), ", ", decimal(imm))
46 }

```

Listing 2. ISA specification basic example (RISC-V)

written using a match expression (see Listing 5 lines 8 to 13). The match statement works analogously. The optional operation annotation is used to assign an instruction to a set of operations. These operation sets can be used to specify the grouping for VLIW architectures or in the MiA section to filter instructions for superscalar microarchitectures.

An instruction encoding assigns fixed values, like operation codes, to certain fields of the instruction word (see Listing 2 line 44). To improve readability, the symbol "'" can be inserted between the digits of a number as demonstrated with the binary number 0b110'0011. With annotations, it is possible to add constraints on format fields to give stronger restrictions on the encoding. On line 43, there is the restriction that the two register indices rs1 and rs2 must be distinct (this is a reasonable constraint but not a requirement in the RISC-V architecture).

An assembly definition specifies how an instruction is represented in a human-readable textual form as used, e.g., by a disassembler or a compiler (see Listing 2 line 45). The keyword `assembly` is followed by one or more names with a common assembly representation. The textual representation is specified by a string expression. The only available string operator is the comma symbol `,` which does string concatenation. VADL offers some built-in string functions: `mnemonic` returns the identifier of the instruction as a string, `register` returns a standard representation of a register based on the name in the register (file) definition. `decimal` or `hex` return their argument in a decimal or hexadecimal string representation.

```

1  format ltype : Inst =                                // immediate instruction format
2  { imm      : Bits<12>                                // [31..20] 12 bit immediate value
3    , rs1    : Index                                  // [19..15] source register index
4    , funct3  : Bits<3>                                // [14..12] 3 bit function code
5    , rd     : Index                                  // [11..7]  destination register index
6    , opcode  : Bits<7>                                // [6..0]   7 bit operation code
7    , immS    = imm as SInt<Size>                     // sign extended immediate value
8  }
9
10 model ltypeInst (name : Id, op : BinOp, funct3 : Bin) : IsaDefs = {
11   instruction $name : ltype = {
12     X(rd) := X(rs1) $op immS
13   }
14   encoding $name = {opcode = 0b001'0011, funct3 = $funct3}
15   assembly $name = (mnemonic, " ", register(rd), ", ", register(rs1), ", ", decimal(imm))
16 }
17
18 $ltypeInst (ADDI ; + ; 0b000) // add immediate instruction
19 $ltypeInst (ANDI ; & ; 0b111) // and immediate instruction
20 $ltypeInst (ORI ; | ; 0b110) // or immediate instruction

```

Listing 3. ISA model definition and instantiation (RISC-V)

instruction, encoding and assembly definitions are often quite similar for different instructions. VADL's macro system helps to reduce redundancies caused by these similarities. As simplicity and safety are crucial requirements for a processor description language, VADL provides a pattern-based syntactical macro system. The core of VADL's macro system are syntax models. Every model has a name, a typed parameter list, a result type, and a body (see Listing 3 line 10). Possible parameter types are syntactic elements like identifiers (Id), binary operator symbols (BinOp), binary constants (Bin), or multiple ISA definition elements (IsaDefs) as used in Listing 3. Further syntactic types for general expressions (Ex), expressions on the left-hand side of an assignment (CallEx), statements (Stat) or encoding elements (Encs) are used in Listing 34 in the appendix. `model` parameters can be used at every position in the body which has the same syntactic type as the parameter. The use of a parameter is indicated by a leading `"$"`. This design decision has two advantages. Firstly, it simplifies parsing as it explicitly marks the use of a macro element. Secondly, the `"$"` captures the model parameter names, preventing name collisions with other ISA definitions. Similar to the parameters, the `"$"` marks the instantiation of defined syntax models. The symbol `" ; "` separates the syntax elements inside an instantiation.

Architectures like the ARM AArch64 or AMD64 ISAs are more complex and have many variants of the same instruction. In VADL, it is required that every variant is specified in a separate instruction definition. For such applications, the core macro system is not sufficient. Therefore, VADL supports higher-order macros (macros which take macros as arguments or which generate a macro), type aliases (e.g. the definition of a higher order macro type),

composition of syntax types (like structures in conventional programming languages), conditional macros, and built-in lexical macro functions (e.g. generating new identifiers). Descriptions regarding these advanced features, their application and their efficient implementation have been presented in an earlier article in full detail [64].

```

1  register file      S: Index -> Word      // general purpose register file , S31 SP
2  [X(31) = 0]        // X31 is zero register ZR
3  alias register file X = S              // general purpose register file , X31 ZR
4  alias register    SP: Word = S(31)      // stack pointer
5  alias register    ZR: Word = X(31)      // zero register

```

Listing 4. ISA register aliasing (AArch64)

In the ARM AArch64 architecture, the register with index 31 of the general purpose register file can serve two different purposes. Depending on the instruction, it can be used as a stack pointer or zero register. The `alias` directive allows access to a register (file) with another name and different constraints, as shown in Listing 4. When using the name `S(31)` or `SP` the real register is accessed; when using the name `X(31)` or `ZR` the zero register is accessed. It is also possible to define an alias of the PC to a certain register of a register file. This is required for the ARM AArch32 architecture.

```

1  enumeration conditions: Bits<4> =      // condition code encodings
2  { EQ                                     // equal      Z == 1
3  , NE                                     // not equal   Z == 0
4  , ..
5  , AL                                     // always
6  }
7
8  function cond2string (condition: Bits<4>) -> String = match condition with
9  { conditions::EQ => "eq" as String      // equal
10 , conditions::NE => "ne" as String      // not equal
11 , ..
12 , _              => "al" as String      // always
13 }

```

Listing 5. ISA enumerations, functions, match and String (AArch32, AArch64)

Listing 5 shows how enumerations are defined and used in VADL. Enumerations are typed. Their values can be either derived implicitly or set in the definition as shown in Listing 22. When using an enumeration value, the full name consisting of the enumeration name and the element name separated by `":"` has to be specified. VADL supports the definition of pure functions. The function body is an expression. Therefore, no statements and side effects are possible. As demonstrated in Listing 5, lines 8 to 13, a `match` expression allows a selection between multiple alternatives and always needs the catch-all condition `"_"` in the last alternative.

VADL has special notations to mark exceptional behavior. In theory, these notations are not necessary, as every exceptional behavior can be described with the basic ISA language constructs. However, neither a human reader nor the compiler generator can distinguish normal behavior from exceptional behavior. Therefore, it is required that exceptional behavior is marked by the keyword `raise` as shown in Listing 6 at line 32. Exception-raising code is often quite similar. Exceptions can be specified similarly to functions to enable code reuse. In contrast to functions, exceptions do not


```

1  instruction set architecture MIPSIV = {
2
3      using IWord      = Bits <32>           // 32 bit instruction word
4      using RWord      = Bits <64>           // 64 bit register word
5      using Address    = RWord              // 64 bit register word
6      using Index      = Bits <5>           // register index for 32 registers
7
8      [next]
9      program counter PC : Address           // PC points to the next following instruction
10     register      EPC : Address            // program pointer
11                                           // saved exception program counter
12
13     [GPR(0) = 0]
14     register file  GPR : Index -> RWord    // zero register
15                                           // general purpose registers
16
17     format R_Type : IWord =               // register 3 operand instruction word
18     { opcode [31..26]                    // operation code
19       , rs   [25..21]                    // 1st source register
20       , rt   [20..16]                    // 2nd source register
21       , rd   [15..11]                    // destination register
22       , shamt [10.. 6]                    // unsigned shift amount
23       , funct [ 5.. 0]                    // function code
24     }
25
26     exception Overflow = {               // overflow exception
27     EPC := PC - 4                        // save exception raising PC
28     PC  := 0xFFFF'FFFF'8000'0180        // set PC to the exception handler address
29     }
30
31     instruction add : R_Type = {          // add with overflow
32     let result, status = VADL::adds(GPR(rs), GPR(rt)) in {
33     if status.overflow then
34     raise Overflow
35     GPR(rd) := result
36     }
37     }
38 }

```

Listing 6. ISA exception handling (simplified MIPS)

return an expression but have side effects caused by assignment statements (see lines 24 to 27). Nevertheless, it must be guaranteed that reads to a register or memory location precede all writes.

To specify exceptional behavior like overflow, the basic VADL built-in functions exist in two flavors. In the normal one, only the primary function result is returned. In the exceptional one, there are two return values: the result and the status (see line 30). The status contains information like overflow, zero, negative, or carry. These built-in functions are used to specify instructions that handle operations with overflow or to specify architectures that have a status register, like the ARM AArch64 or AMD64 architectures.

Listing 7 shows the two variants of the `forall` definition, which enables comfortable specification of tensor operation instructions. Despite the name `forall` this definition is not a loop. It specifies the parallel independent execution of operations on tensors (multidimensional representation of registers). In the example in Listing 7, register file Z is a one-dimensional vector and register file Y is a two-dimensional matrix. Y and Z also could be an alias of register file X. The keyword `forall` is followed by at least one index specifier with a given range after the keyword `in`. The tensor expression creates a tensor with the same dimensions as the provided index ranges. Each resulting element contains the evaluated tensor expression. The comments after the tensor definition in Listing 7 line 10 show this definition's semantically equivalent unrolled version. `fold` is used to specify reductions. The reduction operation defined by the

```

1 instruction set architecture Tensor = {
2   using Index = Bits<4>
3   register file X : Index -> Bits<64>
4   register file Y : Index -> Bits<16><2,2>
5   register file Z : Index -> Bits<16><4>
6
7   format F : Bits<16> = {rs2: Index, rs1: Index, rd: Index, opcode : Bits<4>}
8
9   instruction AddElements : F =
10     Y(rd) := forall i in 0 .. 1, j in 0 .. 1 tensor Y(rs1)(i,j) + Y(rs2)(i,j)
11 //   Y(rd) := ((Y(rs1)(0,0) + Y(rs2)(0,0), Y(rs1)(0,1) + Y(rs2)(0,1)),
12 //             (Y(rs1)(1,0) + Y(rs2)(1,0), Y(rs1)(1,1) + Y(rs2)(1,1)))
13
14   instruction Dot : F =
15     Z(rd) := forall i in 0 .. 3 fold + with Z(rs1)(i) * Z(rs2)(i)
16 //   Z(rd) := ( ((Z(rs1)(0) * Z(rs2)(0)) + (Z(rs1)(1) * Z(rs2)(1))) +
17 //             ((Z(rs1)(2) * Z(rs2)(2)) + (Z(rs1)(3) * Z(rs2)(3))) )
18 }

```

Listing 7. ISA tensor definitions

operator after the keyword `fold` is applied to reduce all results of the expression after the keyword `with` to a single value. Again, the comments after the `fold` definition in Listing 7 line 15 show this definition’s semantically equivalent unrolled version. The `Dot` example shows an instruction which is a building block of a dot product algorithm inspired by the same instruction of the TIC64x architecture. VADL additionally provides constructs for the specification of constant tensors.

VADL also supports language features for the convenient specification of VLIW architectures by applying regular expressions with constraints on operation sets. These features will be presented in a separate article.

3.4 Micro Architecture Section

The microarchitecture section aims to specify the processor implementation at a high level of abstraction. These abstractions enable a concise and understandable specification, as the generators handle many implementation details (e.g., hazard detection or pipeline registers). Users would have more flexibility and control with low level microarchitecture specifications (e.g., in a HDL), but it would be impossible for generators to determine the purpose or the correctness of such specifications. Therefore, only predefined elements configurable by annotations are supported. If these predefined elements are not sufficient, further elements have to be added to the language and the affected generators have to be extended. These extensions to the language have to be done by experts. Their usage should be simple and can be accomplished by inexperienced users.

Firstly, this section will present the core concepts of the MiA modeling view - pipeline stages and instructions. Then, these concepts are illustrated with the example of a 5-stage implementation of the RISC-V architecture. Finally, logic elements that model components outside of the stages (e.g., caches, control logic) are discussed.

3.4.1 Pipeline Stage. Pipeline stages allow users to define the hardware structure of the processor. Each stage defines cyclic behavior, which the processor executes. For example, one processor stage might fetch instructions from memory while another computes arithmetic results. Users can specify the exact behavior using syntax similar to that used to express the instruction behavior in Section 3.3. It is easy to define a concise microarchitecture using powerful language built-ins. Section 3.4.3 provides some examples of pipeline stages.

In addition to the provided examples, annotations can specify a stage’s restart interval and latency period. The restart interval governs the frequency at which new inputs are allowed to enter the stage. In contrast, the latency period controls the number of machine cycles required to complete a single execution. Additionally, users can assign a range to the latency, thus providing pipeline stages of varying lengths.

3.4.2 Instruction Abstraction. The instruction abstraction is a central concept of the MiA. Users can leverage this concept with `Instruction` typed variables. These variables abstract away two dimensions – the kind of instruction and the progress of the instruction execution. The first aspect implies that the MiA specification is not aware of the instructions present in the ISA. Such variables may even represent VLIW bundles. The second aspect implies that the MiA specification is not aware of the execution state. That is, it is not aware of which parts of the instruction semantics have already been computed at any point in the pipeline. The generator resolves these abstractions automatically during the microarchitecture synthesis. If the generator cannot entirely resolve the abstractions, it will raise an error. Section 4.9 explains this process in more detail.

Because the MiA is blissfully unaware of the complexity behind the `Instruction` variable, it can solely interact with the instruction using abstract operations on the variable. For example, it can specify that the instruction should make arithmetic computations using `instr.compute`. VADL provides a set of such operations. We will refer to them as instruction mappings, or simply mappings. Some mappings are very general (e.g., read any register), while others are more specific (e.g., read register file X). This enables users to trade off between precise control and compatibility with other ISAs.

3.4.3 An Exemplary Pipeline. This Section describes the IMPL microarchitecture depicted in Listing 8. A microarchitecture must implement an ISA, such as the RV32I architecture (line 2) in our example. The pipeline consists of five stages. The specifications of each stage will be discussed in the following paragraphs. The `dataBusWidth` annotation determines the width of the memory interface. In this example, reading from and writing to memory is done in 32-bit blocks.

Listing 9 depicts the FETCH and DECODE stages of the pipeline. All stages but the final stage have to specify the result of the stage. The order of stages is defined by accessing the result of a previous stage. The FETCH stage makes use of the `fetchNext` built-in. The result type of this operation (`FetchResult`) abstracts the fetch size while the built-in automatically determines the next program counter. The generator determines the fetch size by analyzing the instructions in the ISA. In the future, VADL users may provide additional options for the fetch operation (e.g., buffers, multiple instructions). To understand the MiA specification, it is sufficient to know that the `fetchNext` built-in loads enough bytes from the correct memory position to represent a single instruction.

The DECODE stage makes use of the `decode` built-in. The primary goal of this built-in is to represent a decoder for the implemented ISA. The generator will synthesize a decoder automatically. It takes a `FetchResult` as input and produces an `Instruction` as output. This is the origin of the instruction abstraction, which was discussed in Section 3.4.2. The `FetchResult` input is obtained from the preceding FETCH stage. Note that the generator can resolve the instruction abstraction because it has access to the ISA. The decoded instruction then reads the source operands from the X register file.

```

1 [dataBusWidth = 32]
2 micro architecture IMPL implements RV32I = {
3   stage FETCH // ...
4   stage DECODE // ...
5   stage EXECUTE // ...
6   stage MEMORY // ...
7   stage WRITEBACK // ...
8 }

```

Listing 8. Execute Stage

```

1 stage FETCH -> ( fr : FetchResult ) = {
2   fr := fetchNext
3 }
4
5 stage DECODE -> ( ir : Instruction ) = {
6   let instr = decode( FETCH.fr ) in {
7     instr.read( @X )
8     ir := instr
9   }
10 }

```

Listing 9. Fetch and Decode Stage

Listing 10 shows the specification for the EXECUTE stage. It is responsible for computing arithmetic operations and executing branches. Firstly, the stage obtains the current instruction from the DECODE stage (line 2). Then, the specification checks whether the instruction is valid (line 3). If not, the stage raises an invalid instruction exception, thus redirecting the control flow to the exception handler (line 4). If the instruction is valid, the stage computes arithmetic operations (line 6) and writes the new program counter (line 8). In addition, the stage verifies whether the instruction is on the correct program execution path (line 7). If this is not the case (branch misprediction), the control logic flushes the EXECUTE stage and all its predecessors. The MEMORY and WRITE_BACK stages in Listing 11 complete the 5-stage pipeline. The displayed definitions define a valid VADL MiA specification.

```

1 stage EXECUTE -> ( ir : Instruction ) = {
2   let instr = DECODE.ir in {
3     if( instr.unknown ) then
4       raise invalid
5     else {
6       instr.compute
7       instr.verify
8       instr.write( @PC )
9     }
10    ir := instr
11  }
12 }

```

Listing 10. Execute Stage

```

1 stage MEMORY -> ( ir : Instruction ) = {
2   let instr = EXECUTE.ir in {
3     instr.write( @MEM )
4     instr.read( @MEM )
5     ir := instr
6   }
7 }
8
9 stage WRITE_BACK = {
10  let instr = MEMORY.ir in {
11    instr.write( @X )
12  }
13 }

```

Listing 11. Memory and Write-Back Stage

3.4.4 Logic Elements. VADL uses the concept of a logic element to model microarchitectural concepts besides stages. The complexity of logic elements varies greatly depending on its semantics. An annotation determines a logic element's type and, thus, its semantics. For example, Listing 12 displays a logic element that allows users to define forwarding paths between stages. The generator must be aware of the logic element's semantics as it must derive the implementation in the microarchitecture synthesis.

Connecting logic elements with the instruction abstraction realizes their full potential. Listing 12 also shows how instructions may read and write values to the previously mentioned forwarding logic. As the generator is aware of the

semantics, it can synthesize the logic of the forwarding network. Furthermore, it can also integrate this knowledge into the hazard detection logic element. After all, the control unit should not stall the pipeline if a forward can resolve the hazard.

```

1 [forwarding]
2 logic bypass
3
4 stage DECODE -> (ir : Instruction) = {
5   let instr = decode( FETCH.fr ) in {
6     instr.readOrForward( @X, @bypass )
7     ir := instr
8   }
9 }

```

Listing 12. Decode Stage with Forwarding Logic

```

1 [ write through ]
2 [ evict roundrobin ]
3 [ entries = 1024 ]
4 [ blocks = 4 ]
5 [ n_set = 2 ]
6 [ attached_to MEM ]
7 cache L1 : VirtualAddress -> Bits<8>

```

Listing 13. Cache Definition

Readers familiar with microarchitecture design may have noticed that the specification does not contain elements for the necessary control logic and hazard detection. If the generator does not find a logic element that handles these circumstances, it inserts a default hazard detection and control element into the MiA. Later, the microarchitecture synthesis determines the necessary control logic for the processor. Letting the generator synthesize these elements changes the role of hardware designers. Instead of testing an idea in a specific processor, they can define it as a new logic element in the VADL generator. Then, they can test this concept in many different configurations with the regular VADL design flow.

3.4.5 Caches. To represent a memory sub-system, VADL defines a cache definition to describe caches. The definition can be parameterized through annotations. Listing 13 defines a cache named L1 with 1024 entries (cache lines). A single cache line has 4 blocks where a single block corresponds to one addressable unit. For instance, this would be eight bits on a byte-addressable architecture. Our cache is defined to be 2-way associative (`n_set`). Since the cache has 1024 entries and each set contains two entries, the cache has a total of 512 sets. Observe that setting `n_set` to 1 is equivalent to a directly mapped cache, while `n_set = entries` makes the cache fully associative. Most importantly, the `attached_to` annotation defines where the cache can fallback to in case of a miss. The fallback storage can be another cache (e.g., level 2), memory or a process. The latter can be used to translate a virtual address to a physical one before accessing main memory for instance. In addition, several behavioral aspects of the cache can be specified, such as write and eviction policy. The attribute naming and design was inspired by [100].

3.4.6 Branch Prediction. In order to model different simple branch prediction schemes and the branch unit behavior, VADL provides the `fetchNext` construct that automatically incorporates branch prediction and control hazard resolving. If no user-defined branch predictor can be found, a default `always_not_taken` branch predictor will be added to the MiA. In general, the architecture of the MiA synthesis is agnostic to the correctness of the branch predictor. This is done by storing the source address alongside the actual instruction variable and comparing the source address to the actual PC at an adequate place. This place is determined by the `instruction.verify` mapping. Listings 8 and 10 show the use of a simple branch prediction scheme. The branch predictor’s implementation can be changed with different annotations. Combining multiple branch prediction schemes is also possible by defining two logic elements and using appropriate instruction mappings on them.

3.4.7 Advanced Techniques. This section will introduce concepts that are required for describing superscalar and out-of-order processors. Many modern processors employ at least one of these two techniques. We will try to introduce these concepts very briefly in the next paragraph. Readers interested in this topic can find more information in [62]. We would like to highlight that we have not yet implemented these constructs in the generators. Therefore, this part of the language is still a work in progress.

Superscalar processors can finish executing multiple instructions per clock cycle. As a result, the overall throughput of the processor may increase. Furthermore, out-of-order processors dynamically schedule the execution of instructions depending on the availability of their inputs. This technique allows the processor to tolerate a certain amount of latency in the instruction stream while still keeping parts of the processor busy.

While these techniques are orthogonal optimizations, i.e., they can be applied separately, we will discuss them on a single example. The following paragraphs will extend the 5-stage pipeline from earlier to a superscalar out-of-order implementation. This MiA employs reservation stations, multiple execution units, register renaming, and a reorder buffer. Again, we will explain these concepts superficially. Interested readers can find more information in [118], [62] and [113].

A superscalar out-of-order processor may be implemented as follows. Firstly, the processor fetches multiple instructions from the memory which are then decoded in parallel. This can be achieved by parameterizing the `fetchNext` and `decode` built-ins from Listing 9. The former one will include the number of bytes to fetch, while the latter includes the maximum number of instructions to decode.

After decoding the instructions, the MiA tracks the instructions in the reorder buffer. This buffer allows the processor to reconstruct the program order after the dynamic dispatching. Listing 14 shows a reorder buffer definition which is modeled with a logic element. The buffer shown is also used to rename the X register file, mapping 32 architectural registers to 64 physical registers (one per reorder buffer entry). This technique is used to eliminate anti- and output dependencies in the original program.

Usually, a superscalar processor contains multiple execution units which are specialized to execute a subset of the supported ISA. However, the processor must ensure that all operand values are available before executing an instruction. To facilitate this, the MiA parks instructions in a *reservation station* until all their operands are ready. Listing 14 depicts two example definitions of a reservation station. The MiA's next task is to dispatch the instructions to their correct reservation station. This step requires separating the instruction stream. For example, only integer instructions must be dispatched to the integer reservation station. Often this is done in a separate stage which we will call DISPATCH.

Listing 15 shows an exemplary definition of a DISPATCH stage. The `filter` built-in is used to partition the instructions into integer instructions and memory instructions. The resulting instructions are then dispatched to the corresponding reservation station. The `used operation` concept models a set of instructions.

```

1 [renames X]
2 [size = 64]
3 [reorder buffer]
4 logic ReorderBuffer
5
6 [size = 16]
7 [reservation station]
8 logic IntegerQueue
9
10 [size = 8]
11 [reservation station]
12 logic MemoryQueue

```

Listing 14. Reorder Buffer and Reservation Station Definition

```

1 operation IntOps = {ADD, ADDI, SUB, ...}
2 operation MemOps = {LW, SW, ...}
3
4 stage DISPATCH = {
5   let is = DECODE_AND_RENAME.ir in
6   let ints = filter(is, @IntOps) in
7   let mems = filter(is, @MemOps) in {
8     IntegerQueue.dispatch(ints)
9     MemoryQueue.dispatch(mems)
10  }
11 }

```

Listing 15. DISPATCH Stage Definition

After dispatching, the instructions reside in the reservation station until all their operands are ready. Execution units subscribe to the reservation stations as consumers, as shown in Listing 16. The example shows two simplified integer units and one memory unit that consume from corresponding reservation stations. Even though both stages use the `i.execute` mapping, the integer units implement arithmetic computations, while the memory unit implements memory access. The VADL generator is responsible for tracking which instructions are scheduled to the respective execution units. Users may also use more specific instruction mappings (e.g., `i.read(@Mem)` in the memory unit) if they prefer.

The resulting instruction streams are then merged in the COMPLETION stage depicted in Listing 17. This is done by using the combine operator (`|`). In the example, the `all` variable joins the three instruction streams together. The processor then marks all instructions in the joined instruction stream as completed in the reorder buffer. Note that this step happens in the order of the instruction execution, not the program order.

Lastly, the MiA must retire the instruction in the reorder buffer. This is done by using the retire mapping as shown in Listing 17. The example shows a possible specification that can retire up to three instructions in a cycle. Once an instruction has retired, its allocated space in the reorder buffer is freed and the architectural state of the processor is updated. Note that contrary to the COMPLETION stage, this process is done in program order. As a result, the original order of the instructions is reconstructed and their architectural side effects are applied in this order.

```

1 stage IntegerExu1 -> (ir: Instruction) = {
2   let i = IntegerQueue.consume in {
3     i.execute
4     ir := i
5   }
6 }
7
8 stage IntegerExu2 = // equal to IntegerExu1
9
10 stage MemoryExu -> (ir: Instruction) = {
11   let i = MemoryQueue.consume in {
12     i.execute
13     ir := i
14   }
15 }

```

Listing 16. Definitions of Execution Units

```

1 stage COMPLETION = {
2   let intExu1 = IntegerExu1.ir in
3   let intExu2 = IntegerExu2.ir in
4   let memExu = MemoryExu.ir in
5   let all = intExu1 | intExu2 | memExu in
6     all.markAsCompleted(@ReorderBuffer)
7 }
8
9 stage RETIRE -> {
10   ReorderBuffer.retire(3)
11 }

```

Listing 17. Retiring Instruction Streams

3.5 Application Binary Interface Section

The ABI ensures consistent and well-defined interoperation between units of object code.

The ABI specification section in VADL supports the definition of

- special purpose registers,
- stack alignments,
- register aliases,
- calling conventions and
- special instruction sequences.

This section provides a description and an example for each of these definitions.

ABI definitions are top-level elements inside a VADL file. The section starts with the keyword `application binary interface` followed by a unique identifier. Since most elements inside the ABI section rely on previously defined ISA elements, it is required to reference an ISA section using the `for` keyword after the identifier. Definitions from the referenced ISA are available inside the ABI section. Listing 18 shows an empty ABI section for the RV32I ISA.

```

1 application binary interface ILP32 for RV32I { }

```

Listing 18. ABI section definition

Specifying the calling convention is one of the most important tasks of the ABI. Calling conventions describe how a function call is executed. The specification contains information on the instructions performing the call, which registers are used to pass arguments or return values, or which registers are managed by the caller or callee. Additionally, it holds information on special-purpose registers, such as a frame pointer, stack pointer, or return address. Figure 20 contains ABI code, that defines a calling convention with special-purpose registers. Each definition has the same

structure, i.e., a descriptive keyword, that declares what register or register group will be specified, followed by a "=" and one or more references pointing to the actual registers. To be more concise, VADL provides a special syntax to address multiple registers with similar names. In the example, the compact expression `a{0..7}` evaluates to `[a0, a1, a2, a3, a4, a5, a6, a7]`. Moreover, Figure 20 showcases the alignment annotation. This is used to specify the stack alignment.

Using expressive names for registers is not only helpful for reading and understanding the specification, but can also have a positive impact on debugging and writing correct specifications. In order to provide registers with additional names, the ABI section provides the `alias register` keyword. With the help of this mechanism, it is possible to assign registers multiple names and use them as a reference throughout the VADL specification. The statement to declare an alternative name for a register follows a structure similar to that of defining special-purpose registers. First, the keywords `alias register` is written, followed by the new identifier. Next, the "=" operator points to the register reference which should be extended by a new name. Note that a single hardware register or register cell is allowed to have multiple different names. If multiple names are available for a specific register, you may use the annotation `[preferred alias]` to emit only the preferred name in the generated code. Listing 19 showcases different alias register statements and enforces the name `fp` for register `X(8)`. In Listing 20, the alias names can be seen in action.

```

1  alias register zero = X(0)
2  alias register ra   = X(1)
3  alias register sp   = X(2)
4  alias register gp   = X(3)
5  alias register tp   = X(4)
6
7  // ...
8
9  [ preferred alias ]
10 alias register fp   = X(8)
11
12 alias register s0   = fp
13 alias register s1   = X(9)
14 alias register a0   = X(10)
15 alias register a1   = X(11)
16 alias register a2   = X(12)
17 alias register a3   = X(13)

```

Listing 19. ABI Register Alias

```

1  [ alignment : Bits <128> ]
2  stack pointer = sp
3
4  return address = ra
5
6  global pointer = gp
7
8  frame pointer = fp
9
10 return value = a{0..1}
11
12 function argument = a{0..7}
13
14 caller saved = [ ra , a{0..7} , t{0..6} ]
15
16 callee saved = [ sp , fp , s{0..11} ]

```

Listing 20. ABI Calling Convention

Finally, the ABI section supports the definition of special instruction sequences. An instruction sequence is a particular order in which a specific list of instructions has to be executed. For example, a call sequence might consist of two separate parts. One instruction loads an address to a specific location and a second instruction jumps to this address and prepares the return register. VADL is able to detect a lot of sequences on its own, e.g., stack manipulations or frame index-related loads and stores. Detecting certain sequences can be challenging due to their explicit inclusion in the processor's ABI or their unreliable detection. To address this issue, the ABI section includes and mandates the use of various sequences such as `call` sequence, `return` sequence, `address` sequence, `nop` sequence and `constant` sequence. Figure 21 defines call and return sequence for the RISC-V processor. Every sequence has a predefined set of parameters with specific meanings. In the case of the presented `call` sequence, the first operand is the target

call address. The body of the definition describes how the address is split into two parts using VADL *modifiers*. The instructions LUI and JALR are then used to load the address into a specific register and jump to it. In addition, the return register X(1) is set. The `call` sequence is appropriately named, as it outlines the steps required to call a specific address or symbol. Similarly, the `return` sequence serves the purpose of returning from a procedure call, as the name implies. Both sequences are only allowed once per ABI section. The `address` sequence definition is used to specify complex address loads. At present, the sequence is designed to load the entire address space and handle only absolute addresses. Additional features are being planned for the `load` sequence to allow for the indication of various types of loads using different flags, such as PC relative or absolute address. For now, the ABI section expects a single `load` sequence. When executing the `nop` sequence, no state transformation should be performed. Finally, the `constant` sequence specifies actions to load constant integers of different sizes. The ABI section supports multiple `nop` and `constant` sequences. All mentioned sequences are analyzed and used by the compiler generator, introduced in Section 4.5, to generate compiler backend source code.

```

1 call sequence( symbol : Address ) =
2 {
3     LUI{ rd = 1, imm20 = hi20( symbol ) }
4     JALR{ rd = 1, rs1 = 1, imm = lo12( symbol ) }
5 }
6
7 return sequence =
8 {
9     JALR{ rs1 = 1, rd = 0, imm = 0 }
10 }

```

Listing 21. ABI Call and Return Sequence

3.6 User Mode Emulation Section

In the field of processor simulation, there is a distinction between two different modes of simulation: User Mode Emulation (UME) (or simulation) and full system emulation (or simulation). In UME, the processor only simulates user mode instructions. System call instructions of the emulated processor are mapped to system calls of the host operating system. In contrast to this, full system emulation also virtualizes system elements of the host architecture, like disks, network interfaces, attached keyboards, or monitors. This means also an operating system has to be executed on the emulated processor to make these virtualized resources available to the emulated processor. VADL currently does not support virtualization of an entire computer but provides language support for convenient UME.

Listing 22 demonstrates the mapping of Linux system calls of an emulated RISC-V processor to the operating system of the host system. The enumeration in lines 1 to 4 defines some Linux system call numbers. The `system call` definition in line 9 specifies that ECALL is a system call instruction, the system call number is passed in register A7, arguments to the system call are passed in registers A0 to A5 and the result of the system call is returned in register A0. Then, similar to the `match` syntax, the required mapping functions are invoked depending on the system call number (lines 10 and 11). The mapping functions are defined by a signature definition and embedded C++ code between the two symbols "`<`" and "`>`" after the keyword `procedure` (lines 14 and 20).

```

1  enumeration SysCall =      // Linux syscall numbers
2  { read      = 63
3    , write   = 64
4  }
5
6  user mode emulation ENV for CPU = {
7
8    // syscall number is in a7, arguments are in a0 .. a5, result is in a0
9    system call @ECALL with format a7 : (a{0..5}) -> a0 = {
10      SysCall::write => write,
11      -              => unserviced
12    }
13
14    procedure write (fd: Word, src: Word, len: Word) -> (res: Word) = -<{
15      std::vector<uint8_t> buf;
16      readMemory(buf, src, len);
17      res = write(getFd (fd), buf.data(), len);
18    }>-
19
20    procedure unserviced (num: Word) = -<{
21      throw std::domain_error("unhandled: " + std::to_string(num));
22    }>-
23  }

```

Listing 22. UME specification (RISC-V ECALL for Linux)

The simulator provides built-in functions like `getFd` and `readMemory`. `getFd` checks if the simulator owns the file descriptor number and returns it. `readMemory` copies `len` bytes from the simulator memory to the specified buffer `buf`. This copying is necessary as the simulator memory is not necessarily contiguous memory, but is implemented as a hash map or an access function to a simulated cache. The simulator generator analyzes the signature of the system call mapping functions and generates code for all necessary register copies. It copies the argument registers to the argument variables, and after execution, the result variable to the result register. This copying code is combined with the embedded C++ code and integrated with the generated simulator.

3.7 Assembly Description Section

To complete the compiler toolchain, a generator tool must be able to create an assembler and a linker so that users can create executable programs for the specified processor. A critical aspect of this task is comprehending the artifacts' inputs and outputs, assembly and object files as well as their interrelation. This understanding must include semantic knowledge, as this is required to establish the relationship between the artifacts. For example, a generated assembler must know how to parse an instruction, associate the string representation with an instruction from the ISA, structure the output object file and emit the binary encoding for the identified instructions in the correct place. Naturally, this knowledge has to be available to the parser generator. Thus, VADL must capture these aspects. This section provides auxiliary information for generating an assembler and linker from a VADL specification.

Fortunately, efforts to define standardized object file formats (e.g., Executable and Linkable Format (ELF)) that can cater to the needs of multiple processor architectures have been fruitful. Such formats dictate the overall structure of the object file while leaving open inherently architecture specific aspects, such as instruction encoding. As a result, processor description languages relying on these formats do not have to capture information regarding the object file's

structure. This restriction reduces the required specification while building on the rich ecosystems that evolve around popular standard formats.

In contrast, assembly languages have no standardized structure like object files. However, many languages are alike. This similarity allows VADL to make some assumptions about the structure of the assembly files to reduce specification effort. Firstly, labels have a predefined syntax: the name followed by a colon (e.g., `loop:`). Secondly, each statement must correspond to a (pseudo) instruction of the processor's architecture. Lastly, the overall structure of the source file is a sequence of labels and statements. A VADL specification thus can solely focus on defining the syntax of the assembly instructions.

Figure 23 presents the structure of an assembly description element, including its three subsections. An assembly description has to refer to an ABI. By extension, the assembly description also depends on the ISA linked to the ABI. The commitment to a particular ABI instead of an ISA is necessary to provide additional information about the usage of some registers. For example, a generated linker could use the defined global pointer to optimize access to certain variables. As with any top-level element, annotations can provide additional information to the generators.

The most crucial element of the assembly description is the grammar definition. It defines the structure of assembly instructions as a formal language grammar augmented with semantic information. For example, users can annotate sub-elements of an instruction with type information, thus capturing the role of an element (e.g., refers to a register). The style of the grammar element is inspired by Xtext [43]. This work will abstain from discussing all intricacies of the grammar element. However, the example in Figure 24 gives readers a good intuition of how the grammar element captures relevant information for the assembler generation. The example shows the definition of a rule that describes RISC-V LUI (load upper immediate) instructions. Register and ImmediateOperand are non-terminals that have a default definition in the language. Users can override these defaults by providing a rule with the corresponding name.

```

1 [ commentString = "#" ]
2 assembly description RV32I_ASM for RV32I_ABI = {
3     directives = {
4         ".word" -> BYTE4
5     }
6
7     modifiers = {
8         "lo" -> RV32I::lo12 ,
9         "hi" -> RV32I::hi20
10    }
11
12    grammar = { ... }
13 }
```

Listing 23. Assembly Description Element

```

1 grammar = {
2     ...
3     // Example: lui ra, %hi(main)
4     LuiInstruction:(
5         mnemonic="lui"@operand
6         rd=Register @operand
7         ","
8         imm20=ImmediateOperand
9     )@instruction
10    ...
11 }
```

Listing 24. Grammar for a RISC-V LUI Instruction

The power of the grammar system is rooted in the type system of the language as it also models the semantic information. Usually, when parsing an assembly file, the algorithm receives tokens with primitive types from the lexical analysis. These tokens do not capture any semantic information. However, an assembler must check whether the tokens satisfy context-dependent criteria. For example, when the assembler encounters an ADD instruction, the first operand has to be a valid register. VADL uses its type system to capture this information. By annotating elements of the grammar with a semantic type, the user instructs the parser generator to insert a conversion routine for the value of the given

element. This routine depends on the input and output types and may include validation and transformation of the input value. For example, the conversion routine from the primitive string type to the register type checks whether a register has a matching name. The procedure’s successful completion asserts that the value refers to a valid register. VADL’s type system conveys this information to other parts of the grammar. A parser can generate a meaningful error message if the validation fails.

Readers may wonder why VADL requires a separate grammar for the assembly syntax even though the ISA section describes assembly formatting functions. The idea is that the language could also define the grammar solely by the inversion of the formatting function. We decided against such an approach for two reasons. Firstly, VADL does not always require grammar specifications for each instruction. By defining conventions for grammar rule names, generators may support users by synthesizing rules from the formatting functions. This approach allows for a graceful degradation of the required amount of specification as users may provide rules on a per-instruction basis. For example, a generator may create the grammar rule from Figure 24 from the associated formatting function. Secondly, if the language relies solely on function inversion, generators must have sophisticated inversion routines, as the system has to support every possible formatting function. By defining the grammar separately, VADL provides an escape hatch if the rule generation capabilities of a generator are not general enough. Lastly, a single assembly instruction may map to multiple valid text representations (e.g., multiple spaces instead of one). This circumstance requires the inversion process to handle alternatives, as the defined language should include all possible representations. Other works addressed this issue by introducing the biased choice operator and special rules for whitespace handling [89]. We decided against relying on this approach, as it significantly increases the complexity of the formatting functions. Reducing the complexity of the ISA section caters to the goal of making computer architecture comprehensible. Understanding the ISA is more important than knowing all possible assembly syntax variations. Thus, making this section more manageable may help VADL users focus on the architecture’s essential aspects.

3.8 Micro Processor Section

The microprocessor modeling view is conceptualized to capture all the remaining aspects of a CPU design to specify the actual composition of the used ISA and ABI which is necessary for generating software simulators as well as generating actual hardware artifacts of a given CPU.

Therefore, this modeling view contains syntax elements to define the start address, the stop condition, the exception handling, startup logic, as well as a firmware section to pre-load or set memory values as well as register states if, for example, the CPU does not operate on a given executable.

Listing 25 depicts a RISC-V CPU specification by using the microprocessor modeling view to define an example processor. This example includes setting up the register state and executing the provided firmware or an external executable. Furthermore, the processor defines exception handling code that the MiA can use. The exception handler saves the current PC in the exception registers and jumps to the exception handler (address stored in `mtvec`). The exception registers are defined in the ISA.

4 IMPLEMENTATION

This section presents the VADL compiler’s implementation aspects, ranging from the compiler overview, language parsing, and domain-specific IR to the detailed descriptions of the code generators for the different PDL artifacts. First, Section 4.1 gives an overview of the compiler’s architecture. Then, each principal component is discussed separately describing and comparing the original VADL and OpenVADL implementation.

```

1 [ target = "rv32i" ]
2 [ description = "32-bit RISC-V Integer" ]
3 micro processor CPU implements RV32IM with ILP32 = {
4
5     start = 0x8000'0000
6
7     stop = PC = 0xe000'0000
8
9     exception invalid = {
10         mepc := PC
11         mtval := PC
12         mcause := 2
13         PC := mtvec
14     }
15
16     startup -> ( ok : Bool ) = {
17         mtvec := 0xe000'0000
18         PC := 0x8000'0000
19
20         ok := PC = start // self-test
21
22         if executable then halt
23
24         firmware // flash
25     }
26
27     firmware = {
28         //
29         MEM<4>( 0x8000'0000 ) := 0b'00000000'00010'00001'000'00100'0110011 // RV32I.ADD
30         // ...
31     }
32 }

```

Listing 25. CPU Specification (RISC-V)

4.1 VADL Compiler Overview

One of the first requirements to the VADL compiler development was that it should be implemented in an efficient and safe programming language. This demanded a strongly, statically typed language which has array bound checking and garbage collection. The original VADL compiler was implemented in Xtend [21]. Xtend is an extension of Java which has type inference and excellent support for string templates. It would have been the perfect language if Xtend – similar to other Java Virtual Machine (JVM) languages – had been implemented as an Xtend to JVM byte code compiler instead of a source to source compiler from Xtend to Java. As line numbers are not related to the original Xtend source code but to the generated Java code, debugging becomes extremely difficult. Therefore, we decided to implement OpenVADL in Java directly using a string template library. Java also has better support in development tools like checking coding style guidelines. This resulted in more effective development and the change of the implementation language proved to be successful.

Figure 1 presents the complete overview of the VADL compiler design. Each processor specification starts as a plain VADL text file. The parser is responsible for reading the VADL text file, applying all macros, and generating an AST. The language compiler handles symbol resolving, type inference, type checking, annotation checking, and constant evaluation. Section 4.2 provides an overview of the parser.

Then, the compiler transforms the AST into an IR, in the original VADL compiler this is the VADL Intermediate Representation (VIR), in OpenVADL this is the VADL Intermediate Architecture Model (VIAM). The IR is the central

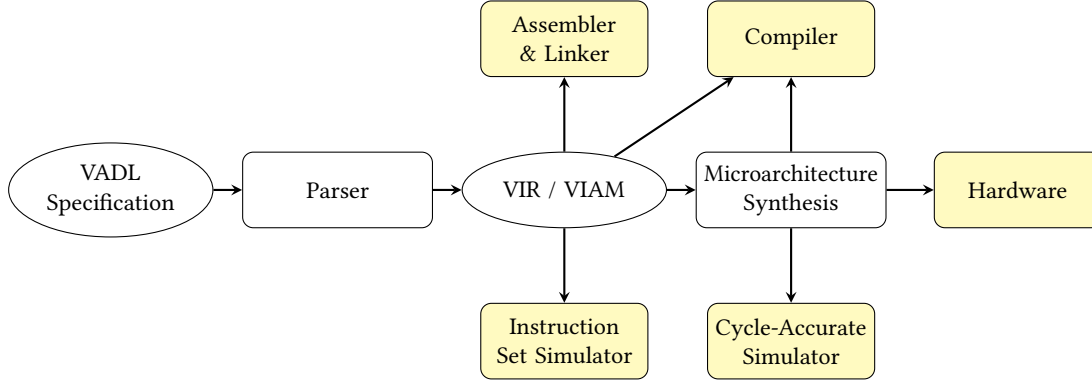


Fig. 1. Overview of the original VADL and OpenVADL Compiler Architecture. Yellow boxes represent generators.

data structure in the compiler, as all generators operate on it. It must be able to describe behavioral aspects (e.g., instruction semantics) and structural aspects (e.g., pipeline stages). After creating the data structure, the compiler does well-known optimizations like removal of redundant operations or dead-code elimination. Section 4.3 describes the VIR in detail. Section 4.4 describes the VIAM in detail.

Generators that do not require knowledge about the microarchitecture can use the IR directly after the transformation. These generators are the assembler and linker generator (see Section 4.6) and the instruction set simulator (for the original VADL’s interpreting simulator see Section 4.7, for OpenVADL’s dynamic binary translating simulator see Section 4.8). Furthermore, the compiler generator can do most of its work without knowing the microarchitecture. However, the generated compiler might perform better at instruction scheduling if the generator has knowledge about the microarchitecture.

The compiler executes the microarchitecture synthesis (see Section 4.9) prior to generators that require microarchitectural details. This step is responsible for integrating ISA and MiA while also bridging much of the semantic gap between VADL and the generated artifacts. As already mentioned, the compiler generator might use this information to tailor the code generation to the processor implementation. Furthermore, the cycle-accurate simulator generator (see Section 4.10) and hardware generator (see Section 4.11) rely on the microarchitecture synthesis.

4.2 Language Parser

4.2.1 Original VADL Parser. VADL’s parser is built on top of the well-established Xtext framework [43]. Xtext is an open-source framework for the rapid development of programming languages and DSLs. The framework takes a grammar file as input and generates a Java-based ANTLR [98] parser, meta-model classes for the syntax tree, and parts of an Eclipse-Modeling-Framework project for effortless Eclipse IDE [66, 117] integration. This work refers to the generated syntax tree consisting of the meta-model classes as Concrete Syntax Tree (CST). To gain more control over the translation and shorten the IDE feedback time, we turned off all non-LL(k) features in the Xtext-generated parser, i.e. *backtracking*, and implemented custom semantic predicates and code actions [99]. The gained context sensitivity is mainly needed to support VADL’s embedded macro language (see Section 4.2.1). After the parsing and macro expansion, the CST is pruned and transformed into the more abstract AST. Please note that the CST contains a lot of syntax-related

information and is primarily used to handle syntactic aspects of an input specification. All further transformations and analyses, e.g., symbol inference or type inference, are performed later on the AST.

The VADL macro language is embedded into VADL. We classify the macro system as a pattern-based and syntactic-typed macro system with the support of higher-order macro patterns [64]. To benefit from the IDE support and check syntactic correctness during parse time, we split the macro system into the *parsing* and the *expansion* phases.

The first phase parses the language and collects information on macro elements. The second phase is a recursive expansion step of the collected macro elements. Since VADL does not perform symbol or type inference on the CST, semantic predicates and code actions interact with a lightweight macro API to compare and update symbol and type information.

A more detailed description of the VADL’s macro system, its types, and implementation can be found in previous work [64].

4.2.2 OpenVADL Parser. The Xtext framework requires to generate a CST which adds additional passes to the parser. Furthermore, the generated parser is based on the Eclipse Modeling Framework preventing ahead-of-time compilation. Therefore, for OpenVADL we replaced Xtext by Coco/R [93, 124]. Coco/R is a pred-LL(1) parser generator based on an attribute grammar. Coco/R does not generate a CST, the generation of an AST has to be specified by the user with attributes. This makes it possible to generate an AST which already has the syntax macros expanded in a single pass. Applying ahead-of-time compilation the OpenVADL parser is up to 150 times more efficient than the original VADL parser. The OpenVADL parser comes with some improvements to the language like macros which generate macros, additional syntax types and operator precedence for expressions. A detailed description of the OpenVADL parser is available in a thesis [95].

4.3 The VADL Intermediate Representation (VIR)

In order to completely decouple specification and code generation while still reusing many intermediate artifacts generated from the VADL specification, we introduced the VIR layer.

This compiler IR is designed to be very close to the abstraction level of a HDL regarding the concepts of expressing sequential and parallel computation logic. Since the current hardware code generator emits Chisel [15] (a RTL abstraction) code, many similar constructs can be found in the VIR.

The goal of the VIR is to provide both structural and behavioral elements well suited to describe the various aspects of CPU design. Because of that, several ideas emerged from existing state-of-the-art IRs for hardware descriptions, like the Low-Level Hardware Description (LLHD) [110] project or the Flexible Intermediate Representation for Register Transfer Level (FIRRTL) [69] project. All of those IRs allow the specification of arbitrary HDL designs. The VIR, however, focuses only on the requirements of CPU designs. Thus, aspects such as register files are first-class elements in the VIR.

Besides the structural elements in the VIR to represent memory components, register states, signals, ports, and overall processor definitions, the main focus of the VIR is expressing data and control flows in the behavioral elements. Two main definitions exist to describe behavior: *functions* and *processes*.

A *function* is a behavioral block in the VIR that describes arithmetic (combinational) functional behavior, which would correspond to a hardware logic computation that can be performed during a single (the same) clock cycle without requiring any memorization of a state. A *process*, on the other hand, describes a state-aware computational behavior that can extend over several clock cycles. LLHD [110] introduced these concepts in their IR design.

In order to represent computations executing within the same clock cycle, one or more VIR *instructions* in SSA form [80] describe linearized operations over virtual registers. One BB represents a computation that can be executed in a single clock cycle. These operations are grouped into a BB and subsequently into a Directed Acyclic Graph (DAG) of BBs to express the data and control flow. One or more BBs can then form single or multi-cycle hardware logic.

Several classical static analysis and transformation passes are implemented on the VIR level, e.g. constant folding, constant propagation, code motion, control flow elimination, inlining, and strength reduction.

Listing 26 shows the VIR representation of the RISC-V ADD instruction. Readers familiar with LLVM [77] or LLHD will see the similarity with the IRs of these projects. Every instruction is implemented as a process. In this case, the process consists of a single basic block. Note how the VIR describes some processor elements (e.g., the register file @RV32.X) as first-class citizens of the IR.

```

1  process @RV32I.ADD.execute (b5 %rs2, b5 %rs1, b5 %rd) -> () = {
2      lbl %bb1:
3          %1 = const u32 4
4          %2 = read b32 @RV32.PC
5          %3 = add u32 %2, %1
6          %4 = probe b5 %rs2
7          %5 = probe b5 %rs1
8          %6 = probe b5 %rd
9          %7 = read b32 @RV32.X, %5
10         %8 = read b32 @RV32.X, %4
11         %9 = cast s32 %7
12         %10 = cast s32 %8
13         %11 = add s32 %9, %10
14         %12 = cast b32 %11
15         write b32 @RV32.X, %6, %12
16         write b32 @RV32.PC, %3
17         halt
18     }
```

Listing 26. VIR Code for the RISC-V ADD Instruction

The example shows several features of the VIR. The syntax is inspired by the LLVM IR. Variables are prefixed with the percent symbol. Names for temporary variables are numbered increasingly. Each VIR instruction is explicitly typed, e.g. the add instruction has the type u32 which represents an unsigned 32 bit integer. Type conversions are done by explicit cast operations, as shown in lines 11 and 12. An example for bit fields are the input parameters, e.g. rs2 which is a bit field of width 5. Registers and register files are accessed with read and write VIR instructions.

The lbl in line 2 specifies the beginning of a BB and is used as a jump target when specifying control flow. Since a VIR process represents a stateful computation with a possible duration of more than a single clock cycle, the probe instruction is used to access stateful variables, in this example the input parameters. Stateful variables can change their value over time. In this example, the probe instructions represents taking the value of the input parameters at a certain point in time. One could think of this as *probing* the input wires.

4.4 The VADL Intermediate Architecture Model (VIAM)

The VIAM serves as an internal representation of a VADL specification used in the OpenVADL project. It is designed to be both generic, ensuring compatibility with various generators, and extensible, allowing each generator to customize it as required. The VIAM is divided into two components: one defines the structural aspects of a VADL specification, including its definitions, while the other captures the behavioral elements, such as instruction behavior.

4.4.1 Behavior Graph. The behavior component of the VIAM is modeled as a multi-graph that integrates both a dependency graph and a control flow graph. The functional nature of VADL’s behavior description enables the construction of a data dependency graph, where data flow is naturally in SSA form due to the semantics of let assignments [10, 20, 114]. Control flow constructs, such as if-else, are represented within the control flow graph. This design is inspired by the IR of the GraalVM compiler [39, 40].

VADL behavior follows a sequential reading style for enhanced readability, while certain constraints enable a parallel interpretation. Semantically, all resource reads complete before instruction execution, and all writes apply only after it finishes. By modeling side effects—such as register writes—as dependencies of end nodes, VIAM enforces this parallel view, discarding execution order and representing only the conditions under which side effects occur. Read operations function as standard expression dependencies, ensuring each register read appears only once in the graph, regardless of multiple occurrences in the source code. Additionally, no explicit ordering is maintained between a read and a write to the same resource, providing generators with greater flexibility in managing side effects.

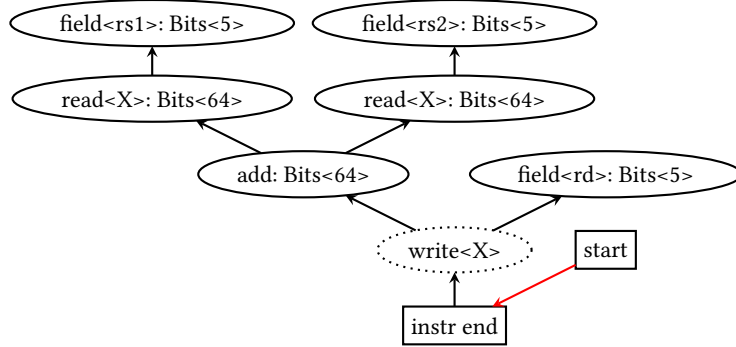


Fig. 2. VIAM behavior graph of the RISC-V ADD instruction

Figure 2 illustrates the behavior graph for the RISC-V ADD instruction. Each instruction begins with a control flow start node and concludes with an end node, representing the minimal control flow structure. The end node depends on the register file write operation, which depends on the instruction’s format field reference *rd*, indicating the destination register index. Additionally, the add operation depends on the register file reads for the operand indices.

To prevent read-write and write-write conflicts caused by the side effect relaxation described above, users are prohibited from writing to the same resource multiple times within a single instruction execution path, as this would lead to undefined behavior. While this constraint can be enforced for registers, it is not always feasible for register files and memory locations, as their exact addresses may not be statically determinable.

4.4.2 VIAM vs. VIR. As described in Section 4.3, the VIR is a quadruple-code IR used in the original VADL implementation, which fundamentally differs from the behavior component of the VIAM. Different generators have distinct

requirements for the IR, depending on how they analyze specifications and apply transformations. During the original VADL implementation, it became evident that the more hardware oriented VIR is not well suited for most generator use cases, leading generators to introduce their own intermediate representations.

The ISS generator produces sequential code that closely resembles the quadruple-code structure of the VIR. However, due to the uniqueness of expressions in VIAM behavior graphs, optimizations such as common sub-expression elimination, copy propagation, and global value numbering occur naturally during graph construction. Furthermore, in the new QEMU-based ISS, dependency analysis and scheduling of potentially conflicting operations are more efficiently managed with VIAM behavior graphs. By lowering the graph to custom control nodes, an operation sequence is generated that can be easily translated into C code.

For the compiler generator, the primary task is to analyze instruction semantics and generate instruction selection patterns. These patterns define the dataflow nodes that must match to emit a machine instruction. In the original VADL implementation, the compiler generator introduced its own dataflow representation, as performing analysis on VIR is unnecessarily complex. In the OpenVADL implementation, this additional representation is no longer required, as the behavior graph itself is directly usable to find selection patterns.

Hardware generation leverages the VIAM to create the Instruction Progress Graph (IPG), which is then combined with the MiA description in MiA synthesis. VIAM behavior design streamlines the process by eliminating the need to construct an intermediate Data Flow Graph (DFG) from quadruple-code and enabling control flow elimination during IPG creation. Unlike the original VIR, where the CAS can execute one basic block per cycle, the DFG in VIAM lacks this strict cycle-based operation grouping. Instead, during VIAM behavior scheduling, dependencies must be analyzed to determine which operations can execute within a single cycle.

The VIR can be easily dumped into human-readable files, which is particularly useful for debugging. In contrast, VIAM behavior graphs cannot be directly represented in a text-based format that is easily readable by developers. To address this, OpenVADL exports the VIAM as an HTML file, embedding all graphs in DOT format. These graphs can then be visualized using an HTML embedded graph viewer, allowing developers to inspect and debug behavior graphs.

4.5 Compiler Generator

This section provides an overview of the design and implementation of the compiler generation component, highlighting its key features and functionality.

4.5.1 Overview. The compiler generation component closes the semantic gap between the high-level ISA specification of instruction semantics and the low-level compiler implementation. Similar to our structure in the VADL tool, modern compilers can usually be split into three main components [77, 116]. A frontend for source-level parsing, an IR for target-independent optimizations and a target-specific backend for target-specific optimizations and generating assembly or bytecode.

One of the most proven approaches for automated compiler generation is to limit the generation process to the target-specific backend, reusing the compiler's parsing and optimization capabilities [14, 128]. By applying this technique, the generated implementation is compatible with state-of-the-art compiler frameworks, enabling us to profit from previous works in compiler research. As a proof of concept, we implemented a VADL compiler backend generator for the well-established LLVM compiler toolchain [77]. In order to keep the support of additional compilers open, we have categorized the compiler backend generation into two subtasks: Extract generic compiler information from the specification and produce compiler backend-specific source files. The Generic Compiler Backend (GCB) component

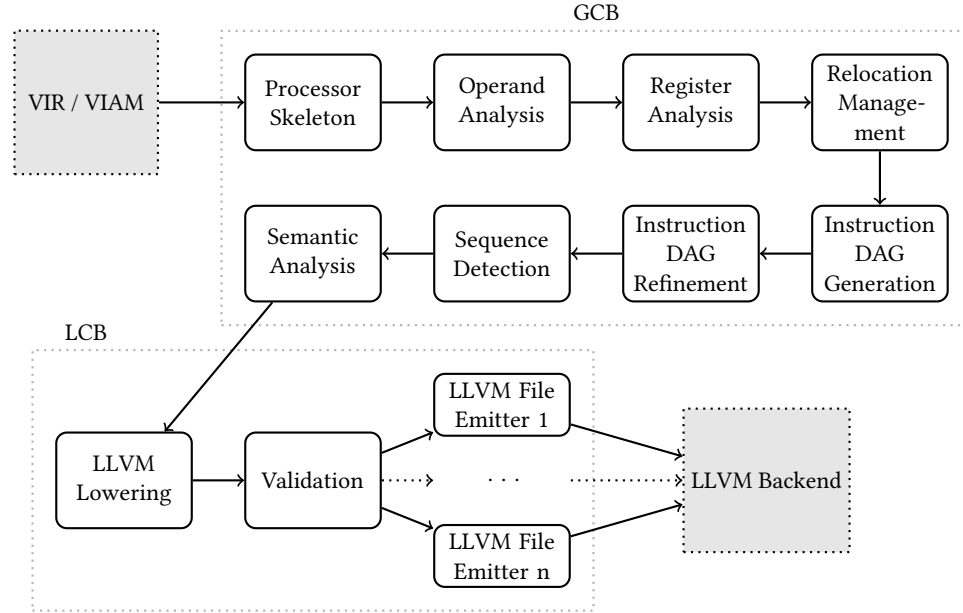


Fig. 3. Compiler Generator Overview

reduces and transforms information provided by the VIR into compiler-generator-relevant information. The created IR, mainly consisting of DAGs, is then passed to a specific compiler toolchain component, producing output files specific to a target compiler's backend. In our case, we implemented the LCB module, responsible for producing a working LLVM backend. Figure 3 gives an overview of the main steps done by the compiler generator component.

4.5.2 Generic Compiler Backend. The GCB module is the core component of the compiler generator. It lifts the VIR entities to a new abstraction, only retaining information relevant to the compiler model. The resulting intermediate representation is the basis for further compiler synthesis steps. While we mainly focused on generating an LLVM backend, the GCB IR could be extended to support a variety of compiler backend targets.

The GCB IR acts as a further abstraction layer over necessary compiler elements. Introduced abstractions mostly behave like glue code between VIR, C++ sources and newly collected or synthesized information. At the beginning, the GCB generation starts by bundling the low-level VIR and generated C++ source units for relocations and immediate encoding, decoding and predicate functions into high-level compiler elements. During this first step, most of the core structure of the GCB model is created. The generated model can be seen as a processor skeleton extended during the execution of the GCB passes. All further passes mainly deal with analyzing instruction semantics.

Next, the dynamic format fields are examined to recognize the instruction operands and assign them to a specific type. The preparatory work in the VIR is crucial here, as it minimizes the semantics and simplifies the recognition of register accesses or immediates used as addresses or arithmetic operands. The categories used for instruction operands are *register-* or *immediate-*operands. Constant register class access, e.g., $X(\emptyset)$, single register access, or the use of register values or immediates as memory addresses are all managed inside the instruction behavior and have no impact on the operand type. The only additional distinction is, if the operand is used as input or output operand. In contrast to

the LLVM specification language *TableGen*, VADL is able to work with multiple input and output operands. To deal with these shortcomings of target backends, the GCB is able to transform most instructions into a suitable form by duplicating operands that occur as input and output operands, and generating additional instruction operand constraints. After the operands are collected, an additional analysis flags immediate operands that are used as relative and absolute jump addresses, respectively.

Furthermore, the GCB models all kinds of register-related elements as *register resource*. First, a distinction is made between single hardware registers and register classes. While a single hardware register only contains a VIR type, a register class is a set of hardware registers. Second, the register classes are separated into *hardware* register classes and *virtual* register classes. A hardware register class must provide registers for each given index. A virtual register class, on the other hand, is a modification of a hardware register, modeling constraints and slight modifications, e.g., replacing a single register with a zero register or restricting specific indices. This becomes useful as some hardware instructions that access register files have particular behaviors for specific index values. The VADL specification may use the `alias register files` mechanism to restrict or modify the access of register files. To model this behavior, the GCB analyzes these artificial resources, collects information on the different indices, and creates *virtual register classes* for the affected instruction operands. Since single hardware registers are not viewed as operands, they need special attention. A separate register analysis traverses the instruction behavior and marks the single registers used for each instruction individually. The information gained is helpful for instruction selection in the backend.

The VADL tool automatically creates a relocation symbol and function for every specified modifier relocation. However, this representation usually needs to be more high-level. The GCB creates specific low-level relocation behavior based on the instruction's immediate operands to use relocations during linking. This process looks at every instruction separately, but future work to combine instructions with similar formats into bundles is already planned. After the relocation management step, the compiler backend has information on modifying the bit-fields of encoded instructions to perform specific relocations.

Moreover, a significant transformation done by the GCB is converting the instruction semantic to a DAG form. Alternatively, we experimented with keeping the VIR representation, which turned out to be unnecessarily complex as most of the applied analyses, transformations, and especially pattern-matching tasks are better suited for DAGs. In an iterative process, the initially rudimentary graph nodes are merged into more complex node patterns. This process serves a dual purpose. Firstly, it expedites the identification of significant patterns in the various ABI sequences, and secondly, it guarantees a more concise representation of the instruction semantic. The implemented DAG node kinds are inspired by the LLVM *TableGen* nodes. The reason for this is that *TableGen* is a well-developed language and secondly, it shortens the development to generate an LLVM-compliant backend. This decision does not impact the generality of the GCB.

The LLVM backend requires C++ helper functions that produce specific sequences of instructions to function correctly. These sequences are responsible for copying registers, loading memory addresses, dealing with complex immediates, or performing memory offset calculations. LLVM does not deduce these sequences from the provided patterns. Since statically retrieving this information from the generated *TableGen* patterns is impractical, VADL additionally performs a simple instruction selection for the mentioned sequences. Most of the C++ helper functions are also relevant for particular ABI-specific behavior. In contrast to a simple value move, calling conventions or more complex loads with symbols cannot be derived automatically. C++ code which deals with more complex or ABI-related sequences is synthesized using information from the VADL ABI section. See Section 3.5 for more details.

Finally, all DAG patterns are checked for semantically equivalent alternative forms. A separate pass performs semantic preserving transformations and stores the newly generated patterns to their original instruction. This step is beneficial to achieve more excellent coverage of necessary comparison patterns as the actual hardware usually only provides the minimal complete set of compare operations.

This concludes the generation of a general processor model, which is passed to a specific backend generator.

4.5.3 LLVM Compiler Backend. The LCB starts by applying a lowering, followed by a validation pass on the received generic processor model.

The lowering step transforms the generic model into a state where it only needs to be output by the emitters. First, generated C++ classes and functions are adapted to be compliant with the LLVM infrastructure i.e., modification of types and signatures. Second, the lowering pass tries to legalize the generated patterns. This primarily consists of casting immediate operand types to a suitable size of a power of two and forcing a uniform operation bit-width for generated instruction patterns. Finally, it removes incomplete or irrelevant information that LLVM or *TableGen* cannot process.

Since the lowering process modifies and removes information, the LCB needs to validate the final processor model. Currently, the validation consists of ensuring the existence of LLVM essential sequences, specific purpose registers or ABI-relevant information to successfully compile simple test programs.

The remaining part of the LCB consists of individual emitters and templates for each LLVM source file. This enables us to locate files and adapt their content quickly if needed. After the lowering step, the processor model is no longer transformed or modified. All needed information is contained inside the model and is queried through the different emitting strategies.

Finally, the backend structure generated by the LCB is designed to be copied over an existing LLVM project. The LCB generates a configuration script for convenience, which can be used to move the generated backend and compile the LLVM project with suitable settings.

4.5.4 LLVM Compiler Backend - Instruction Matching. Listing 27 shows a snippet from a VADL specification which contains the ADD instruction. It reads the values from two registers, adds them together and writes the result into register rd. LCB's central task is to provide the LLVM backend with the tree patterns representing the semantics of various instructions specified in the VADL specification. During compilation, LLVM uses these patterns to completely cover a program's dataflow representation. This phase is called *Instruction Selection* and LLVM's dataflow representation is a directed acyclic graph which is called *Instruction Selection Graph*. Once LLVM has found a complete cover of all the nodes from the *Instruction Selection Graph*, it will emit machine instructions which have been specified in the VADL specification. Thus, the generated code will be semantically equivalent to the LLVM IR program. LCB performs the following steps to generate the tree patterns for the *Instruction Selection*:

- Convert the VADL specification into VIR representation
- Extract the semantics from the VIR by constructing a dataflow graph
- GCB matches known patterns to recognize instructions based on their semantics
- LCB converts the dataflow graphs into *TableGen* records used by LLVM

In our example, the code from Listing 27 is converted into VIR which you can find in Listing 28. Next, the VIR is converted into a dataflow graph which captures the semantics, as depicted in Figure 4. Lastly, LCB emits the mappings as *TableGen* definition, shown in Listing 29. Note that it is not always possible to pattern match the semantics of an

instruction. *TableGen* does not support instructions with multiple results. So whenever the semantic representation is not a tree, mappings cannot be emitted.

```

1 instruction ADD : Rtype =
2 {
3   X(rd) := ((X(rs1) as SInt) + (rs2)) as Bits
4 }
5 ...

```

Listing 27. VADL ADD instruction

```

1 process @RV32I.ADD.execute.gcb
2   (b5 %rs2, b5 %rs1, b5 %rd) -> () = {
3     lbl %bb819:
4     %6372 = probe b5 %rs2
5     %6373 = probe b5 %rs1
6     %6374 = probe b5 %rd
7     %6375 = read b32 @RV32.X, %6373
8     %6376 = read b32 @RV32.X, %6372
9     %6377 = cast s32 %6375
10    %6378 = cast s32 %6376
11    %6379 = add s32 %6377, %6378
12    %6380 = cast b32 %6379
13    write b32 @RV32.X, %6374, %6380
14    halt
15  }

```

Listing 28. ADD instruction's VIR

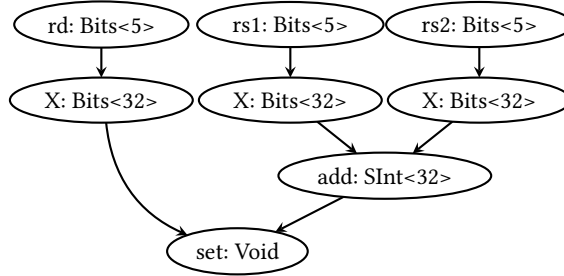


Fig. 4. ADD instruction's semantics

```

1 def ADD : Instruction
2 {
3   ...
4   let OutOperandList = (outs X:$rd);
5   let InOperandList = (ins X:$rs1, X:$rs2);
6   ...
7 }
8
9 def : Pat<(add X:$rs1, X:$rs2),
10   (ADD X:$rs1, X:$rs2)>;

```

Listing 29. ADD instruction's TableGen

4.5.5 Changes with OpenVADL. The previous sections discussed the original compiler generator. The design changes in OpenVADL's frontend have led to changes in the new compiler generator. This section discusses two major changes.

The first major change is the underlying IR. The original VADL uses the VIR, which is a quadruple code describing the machine instruction's behavior. The original GCB creates a dataflow representation from the VIR to generate the TableGen patterns for instruction selection. The OpenVADL implementation uses the VIAM as IR which already contains the dataflow representation. So, OpenVADL's GCB does not need the additional analysis and translation step.

The second major change is the heuristic labeling of pseudo instructions and machine instructions. For constant materialization, frame setup and frame destruction, LLVM needs a certain set of instructions to be present. LCB checks if all of those instructions are defined in the VADL specification, and labels them appropriately. LCB finds those instructions by heuristically analyzing their behavior. E.g., to identify an `ADD`, the heuristic labeling checks whether there is a `WriteRegFileNode` with an addition `BuiltinNode` which has two `ReadRegFileNode` as input. An example of such an instruction specification and its VIAM representation can be seen in Figure 2. Another benefit of the improved labeling is that it simplifies the lowering. By labeling instructions, OpenVADL’s LCB can group them together and apply different lowering strategies to generate a TableGen pattern. For example, the lowering of arithmetic or logical instructions is handled differently than the lowering of jump instructions.

4.6 Assembler and Linker Generator

This section discusses the assembler generation within the original VADL LCB prototype. Its task is to emit the assembler and disassembler components of the generated backend. This work abstains from outlining the exact architecture of the generated artifacts because the LLVM infrastructure dictates large portions of the design. Interested readers may find additional information in the official LLVM documentation¹. Instead, we will cover a set of generic components necessary for a full-fledged compiler toolchain. The text will focus on how the VADL tooling can extract the required information from the specification. In addition, it introduces a straightforward approach to generating grammar rules from the assembly formatting function. Lastly, this section elaborates on handling relocations at the boundary between the assembler and linker.

As discussed in Section 2, a native program has two important persistent representations - assembly and object code. Each tool operates differently on these file types. For example, an assembler must parse an assembly file and produce an object file. In addition to the persistent manifestation, the tools use internal data structures during processing. Figure 5 illustrates the transformations between the representations and the responsible LLVM components. The following paragraphs discuss the depicted components briefly.

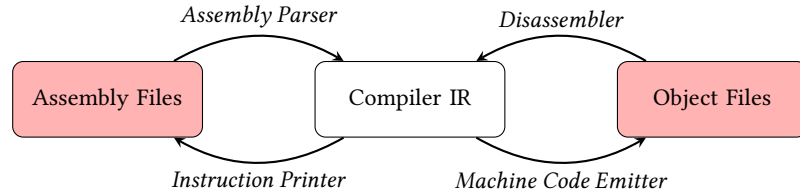


Fig. 5. Overview of Generated Components and Their Inputs and Outputs. Red Boxes Denote External Representations.

4.6.1 Instruction Printer. This component is responsible for transforming the internal representation into assembly text. The compiler uses this component to emit assembly files. Furthermore, the disassembler uses it to print the decoded instructions to a command line interface. Implementing this functionality requires knowing how to express an instruction as text. VADL captures this relation with the assembly printing functions in the ISA section. The VADL tool uses the regular translation path via an implemented C++ code generator to obtain an implementation for each instruction type.

¹<https://llvm.org/docs/>

4.6.2 Assembly Parser. The inverse to printing is parsing the assembly text into an internal representation. The assembly parser implements this transformation. The assembly description element is the primary information source for this task. Section 3.7 introduced this definition. Generating a parser from a formal grammar is a well-studied problem. Interested readers can find an excellent introduction in [34, Chapter 3]. The VADL tool generates an LL(1) recursive-descent parser from the grammar specification. These operations include recording, transforming, and validating values extracted from the text. Section 5.6 discusses some limitations of this parser implementation in the context of assembly languages.

After parsing, the algorithm identifies a set of named operands. Then, it compares the name and content of these operands to the instructions provided by the ISA. For example, matching a RISC-V ADD instruction requires mnemonic, rd, rs1, and rs2 operands. Furthermore, the mnemonic operand must equal "add". After finding a match, the parser instantiates the corresponding internal representation. If the algorithm finds no matching instruction, the tool reports an error to the user. The grammar validation ensures that the operand names match with at least one instruction. However, this validation does not reason about an operand's content, thus it is not guaranteed that a corresponding instruction can be found. Lastly, the program assures further invariants. For example, it asserts that a constant's value does not exceed its range in the matched instruction. During this process, the parser applies the necessary immediate decoding functions defined in the format. Determining the transformation functions is straightforward because the parser knows the operand name and the instruction type.

4.6.3 Disassembler. The central task in generating the disassembler, apart from understanding the object file format, is decoding the instructions into the internal representation. The instruction format and constant format fields define the decoding function. LLVM allows defining this information in a TableGen file. From this, the infrastructure can automatically synthesize the decoder. Of course, a VADL generator could also synthesize this functionality without LLVM from the same information.

4.6.4 Machine Code Emitter & Linker. The machine code emitter encodes the internal representation in an object file format. This task involves encoding instructions and recording metadata. LLVM can synthesize the encoding function from the TableGen file. In addition, the final object file must include relocation entries. This information is necessary to convey program details to the subsequent linkage step. It is essential to highlight that this information is necessary for using symbols in assembly (e.g., function names). Most importantly, a relocation entry contains a type and a symbol name. The relocation type entails information on how the linker shall resolve the symbol (e.g., relative or absolute). For example, one relocation type could describe the usage of a symbol that is resolved relative to the current instruction (e.g., RISC-V branches).

Before the assembler can record relocation definitions, the assembler and linker must agree on the supported relocation types. Naturally, the VADL generator emits declarations for the relocations from the ISA section. In addition, the tool synthesizes generic relocations for immediate format fields. The latter type is required so that users do not have to define a relocation that applies no transformation to the value. For example, the relative RISC-V branching instructions use this feature in our processor description.

The biggest concern when generating the *linker* is understanding the object file format. In the LCB, the LLVM infrastructure provides this capability. The architecture-specific code focuses on applying relocations to the encoded instructions.

4.6.5 Grammar Inference. Before generating the components mentioned above, the VADL tooling infers grammar rules based on formatting functions from the ISA. The problem of synthesizing a formal grammar from a pretty printer

is related to program inversion, as the grammar defines the inverse operation. The function's parameters are the instruction's operands. The result of each formatting function is a plain assembly string. As a result, the inverted function computes the operands from the assembly string. Our implementation combines multiple ideas from program inversion to leverage this relationship.

The first observation is that, given an interpreter for VIR functions, an algorithm can synthesize an inverter by trying all possible input combinations and recording their output. This result captures a unique input-output mapping if the pretty printer is injective, i.e., the computed output values are unique. The program could obtain a formal grammar by generating an alternative over the outputs from the mapping. Each choice is augmented with the initial input values, resulting in an inversed mapping from output to input values. However, this becomes impractical as the input domain size can increase quickly. In addition, the grammar structure resulting from this approach is ill-suited for many critical aspects of a parser. Essentially, the grammar boils down to expressions that check if the input text matches precisely with a particular string, such as "add x1, x2, x3", and then assign specific values to the corresponding variables. Therefore, grammar rules no longer contain structural information. This information is crucial for tasks like automatic error message generation.

Another approach is synthesizing the grammar rule from the VIR function by defining additional grammar generation semantics for each instruction. This approach results in a well-structured grammar and can handle large input domains as the algorithm does not have to interpret all possible values. However, once control structures are involved, writing a general inversion algorithm can take time and effort. The primary reason is that the inverter must be able to handle all VIR instructions used in the formatting functions. In addition, the inverter must consider interactions between multiple instructions. For example, if the formatting function uses multiple conditional constructs with the same selection input.

The VADL tooling uses a hybrid approach to remedy the problems of both techniques. It directly handles widely used VIR instructions, such as string concatenation. Once the algorithm encounters a VIR instruction that it cannot directly process, it switches to an interpretation-based grammar inference technique. Implementing this approach allows leveraging synergies with other components that require an interpreter. The remaining puzzle piece for a functioning parser generator is the lexical analysis, which is responsible for tokenizing the input text. VADL defines a set of built-in terminal symbols that the generator maps to equivalent LLVM token types. By not allowing users to specify custom terminal rules, the system can reuse the LLVM tokenizer without modification. Interested readers can find an excellent introduction to lexical analysis in [34, Chapter 2]. A detailed description of our assembler generator can be found in [112].

4.7 Instruction Set Simulator Generator

The ISS of VADL is a functional instruction set simulator only. It does not emulate non functional behavior like caches as the CAS does. The design space for implementing an ISS offers a vast number of options. Because of limited resources we decided to go for a simple and generic but efficient simulator. Therefore, an ISS using JIT technology was out of scope. Instead, we opted for an implementation based on efficient interpretation. The fastest interpretation technique available is Direct Threaded Code (DTC) where the instruction memory only contains pointers to the code which emulates the instruction. For the simulation of von Neumann architectures, DTC requires an additional instruction memory mirroring the data memory. Depending on the instruction size and the size of pointers, this instruction memory would have a multiple of the size of the data memory. Furthermore, most entries of the instruction memory would be empty and the initialization overhead of these empty entries would be huge. Therefore, the ISS employs a hashmap where an instruction memory address is mapped to a pair comprising of a pointer to the instruction's emulation code

and the instruction at that address. This design also eliminates a range check for the instruction pointer as only valid addresses are entered into the hashmap. When it is necessary to simulate self modifying code there are two possibilities: It can be checked if the returned instruction is equal to the instruction in the data memory. Or it can be checked at every write to the data memory, if the write invalidates an entry in the hashmap. VADL's ISS uses the first checking technique.

```

1 format ltype : Word =      // ltype
2   { imm      : Bits<12>    // [31..20]
3   , rs1      : Index       // [19..15]
4   , funct3   : Bits<3>     // [14..12]
5   , rd       : Index       // [11..7]
6   , opcode   : Bits<7>     // [6..0]
7   , immS     = imm as SIntR
8   }
9
10 instruction ADDI : ltype = {
11   X(rd) := ((X(rs1) as SInt) + immS) as Bits
12 }
```

Listing 30. ADDI instruction definition in VADL

```

1 typedef unsigned int sint32;
2 typedef unsigned int uint32;
3
4 sint32 X[32];
5
6 inline uint32 ADDI (const uint32 PC,
7                   const uint32 instr) {
8   uint32 rd  = (instr >> 7) & 0x1f;
9   uint32 rs1 = (instr >> 15) & 0x1f;
10  sint32 immS = (sint32) instr >> 20;
11  X[rd] = X[rs1] + immS;
12  return PC + 4;
13 }
```

Listing 31. ADDI translated to C++

In the ISS an instruction specification is represented as an inline C++ function which takes the program counter and the instruction word as arguments and returns the updated program counter. Simple encoded format fields are derived via shifting and masking. Complex encoded format fields can be predecoded and additionally stored in the hashmap and only a pointer to these elements is passed to the C++ function. The presented translation in Listing 31 is simplified. Because of the transformations, casts and optimization on the VIR the generated code only contains assignments with a single binary expression and mangled names. The C++ compiler optimizes and simplifies the expressions in the generated code. Therefore, the ISS generator only has to apply a few optimizations during C++ code generation. The ISS main interpreter loop consists only of a single access to the hashmap (which returns the address of the label where an invocation of the inlined translated function has been positioned) and a jump to that address.

The generation of the C++ code is straightforward. There is just a simple analysis of the assignment to and the use of the program counter to add the correct program counter updating code. The decoder is already available in the VIR and can be reused. It is combined with the function which adds new elements to the hashmap on a miss in the map.

To facilitate the validation of the generated simulators and of processor specifications, the simulator supports trace generation and co-simulation. The amount of checked execution state can be controlled by command line options. Co-simulation has been done against other simulators and real hardware (RISC-V, AArch64). The validations have been conducted using existing processor validation suites. A detailed description of the simulator generator and an evaluation of performance and co-simulation can be found in a thesis [90].

4.8 QEMU Generator

OpenVADL introduces a new ISS generator to overcome the limited performance of VADL's DTC based ISS. The new ISS is based on QEMU, an open-source emulator and virtualizer that enables hardware virtualization and full-system emulation for various architectures.

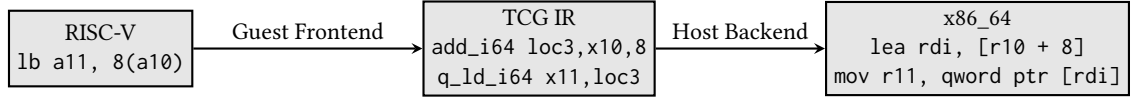


Fig. 6. TCG Translation Process

QEMU enables programs compiled for a guest architecture to run on a different host architecture using dynamic binary translation for high performance. Its modular design simplifies the addition of new guest and host architectures. To decouple guest and host implementations, QEMU employs the Tiny Code Generator (TCG). As shown in Figure 6, the guest frontend reads and decodes instructions from a Translation Block (TB) (a basic block of target code) and translates them into TCG ops, QEMU's architecture-independent IR. The TCG then optimizes this IR before passing it to the host backend, which translates it into machine instructions.

To generate a minimal QEMU target, the generator must define four key components: a *CPU state*, which stores all register values and CPU-related states; a *machine*, responsible for memory initialization and firmware loading; an *instruction decoder*; and the *TCG translation*. The CPU definition can be generated straightforwardly, as it is directly derivable from the VIAM.

The machine definition is relatively generic, incorporating only memory definitions from the VADL ISA specification. When defining the microprocessor, which serves as the entry point for ISS generation, users can annotate it with `[enable htif]`. This enables support for the Berkeley Host-Target Interface (HTIF), a simple protocol used in the RISC-V Spike simulator to facilitate communication with the simulation host. HTIF operates by mapping user-defined memory addresses to callbacks that interpret commands sent by the simulated program. For example, this mechanism allows to exit from full-system emulation with a specific exit code, which is particularly useful for running self-verifying tests on the ISS.

QEMU provides its own decode tree format, allowing frontend developers to define a readable instruction format specification, similar to what the VADL language offers. The build system then generates C functions and structs, which the frontend uses to decode and process instructions. While the VADL Decode Tree (VDT) generator can emit this format, the format itself lacks support for variable-length instructions. To overcome this limitation, the ISS generates its own C-based decoder instead.

The core functionality of the generated TCG frontend lies in TCG operation generation. The ISS generator produces a translate function for each instruction, which generates a sequence of TCG operations. These operations work on strongly-typed variables, where each instruction follows a fixed format: a number of leading output variable operands, followed by input or constant variables. Examples of such translate functions can be seen in Listing 32 and Listing 33, which show implementations for the ADDI and BEQ RISC-V instructions, respectively.

There are different types of variables in TCG, defined by their lifespan and modifiability. Global variables persist across all TBs and correspond to memory in the CPU state. For example, a register in the CPU state is represented as a global variable, and any write to the TCG variable during instruction execution is propagated to the corresponding register. Constant variables exist throughout a TB but are immutable singletons. They are allocated on demand during translation, only if a constant for the given value does not already exist. Temporary TB variables live for the duration of a TB but are discarded upon any exit.

The following sections outline the key passes of the ISS generator, presented in the order of their execution, which are essential for generating TCG translation functions.

```

1 bool trans_addi(
2     DisasContext *ctx,
3     arg_addi *a)
4 {
5     TCGv_i64 x_rs1 = get_x(ctx, a->rs1);
6     TCGv_i64 x_rd = dest_x(ctx, a->rd);
7     TCGv_i64 const_immS =
8         tcg_constant_i64(a->immS);
9
10    tcg_add_i64(x_rd, x_rs1, const_immS);
11    return true;
12 }

```

Listing 32. QEMU translate function of ADDI instruction

```

1 bool trans_beq(
2     DisasContext *ctx,
3     arg_beq *a) {
4     TCGv_i64 x_rs1 = get_x(ctx, a->rs1);
5     TCGv_i64 x_rs2 = get_x(ctx, a->rs2);
6     TCGLabel *l_else_0 = gen_new_label();
7     tcg_brcond_i64(TCG_COND_EQ, x_rs1,
8                   x_rs2, l_else_0);
9     gen_goto_tb(ctx, 1,
10                ctx->base.pc_next + a->immS);
11    gen_set_label(l_else_0);
12    ctx->base.is_jmp = DISAS_CHAIN;
13    return true;
14 }

```

Listing 33. QEMU translate function of BEQ instruction

4.8.1 Operation Decomposition. While the VADL specification allows arbitrary bit widths, QEMU imposes a 64-bit limit for most operations. This becomes problematic when an instruction specification requires types larger than 64 bits. For example, the MULH instruction in the RV64IM specification performs a long multiplication of two 64-bit values and extracts the upper half of the 128-bit result. To handle such cases, the Operation Decomposition pass splits these operations into multiple logically equivalent operations that only accept and return values with a maximum size of 64 bits.

4.8.2 Side Effect Scheduling. As discussed in section 4.4, the VIAM behavior graph represents expressions and side effects using a dependency graph. However, since TCG ops execute sequentially, this dependency graph must be scheduled. The first step in this process is scheduling side effects, such as register writes. Additionally, the pass analyzes whether a side effect causes an instruction exit by modifying the program counter. Non-exit side effects are scheduled at the start of the control flow branch, while program counter manipulations are placed immediately before the branch end. This ensures that a jump out of the instruction does not occur before all other side effects have been applied.

4.8.3 Safe Resource Read. To ensure that writes do not occur before reads to the same resource, potentially conflicting reads must be scheduled before any writes to that resource. Since register file indices and memory addresses are not statically known, all reads to these resources must be conservatively treated as potential conflicts with all writes to the same resource.

4.8.4 TCG Expression Scheduling. Before lowering to TCG operations, it must be determined which expressions are evaluated during TCG translation and which are executed at runtime when the translated TCG code is executed. Expressions evaluated at runtime must be converted into TCG operations and scheduled accordingly. This includes all expressions that depend on the CPU state or memory, such as register reads.

Conversely, expressions that depend only on immediate values, such as format field values, can be computed at translation time and represented as constant TCG variables. These expressions do not require scheduling, as their dependencies can be directly translated into C expressions.

After this pass, all dependency nodes corresponding to TCG operations are correctly scheduled.

4.8.5 TCG Branch Lowering. At this stage, branches within the instruction are represented in the control flow using if-else nodes. However, TCG implements jumps within a TB using goto-like operations, such as set_label, br and

brcond. This pass analyzes which if-else control flow must be converted into TCG operations—specifically, those where the condition expression was previously scheduled as a TCG operation. These control flow structures are then transformed into a linear sequence of TCG operations using labels and conditional branching.

If-else control flow that do not require TCG translation remain unchanged and are later directly translated into if statements in C.

4.8.6 TCG Op Lowering. The scheduled dependency nodes are lowered into control nodes, each corresponding to one or more TCG operations. During this process, a node retrieves its destination and input TCG variables from a context that generates variables on demand and attaches them to the dependency node. Once lowering is complete, all dependency nodes are removed from the graph. The resulting structure is a Control Flow Graph (CFG) consisting of TCG op nodes in SSA form.

4.8.7 TCG Variable Allocation. To minimize the number of temporary TCG variables, a variable allocation pass is applied to the TCG CFG. First, the live ranges of previously created temporary variables are determined. Then, graph coloring is used to compute an optimized TCG variable assignment. The primary objective is to maximize the reuse of written registers, reducing unnecessary temporary allocations.

4.8.8 Putting It All Together. After the final pass before code generation, the instruction behavior graphs directly reflect the structure of the C code to be generated. TCG operations correspond to control nodes with a single successor, if-nodes translate to C if statements, and expression nodes map to C expressions. A code generator processes this graph, producing a C function named `trans_<mnemonic>`, which takes a TCG context and a struct containing all format fields of the instruction. This translation function is then invoked by the decoder during QEMU execution.

4.9 Microarchitecture Synthesis

The Microarchitecture Synthesis is an intermediate step executed before obtaining a cycle-accurate simulator or a hardware schematic. Extracting this step is sensible because both artifacts require identical analysis and transformations. After all, the cycle-accurate simulator shall be able to emulate the hardware implementation. Before generating an artifact, the compiler must lower the high-level microarchitecture to standard VIR processes. I.e. the input to the microarchitecture synthesis are the ISA specification and the MiA specification. The output is a program in the VIR 4.3 representing the unification of both the ISA and the MiA. This endeavor currently consists of six major tasks:

- (1) By splitting instructions into parts the compiler maps the instruction semantics of the ISA to the placeholders in the MiA. The system then replaces the placeholders with the corresponding parts of the instruction semantics.
- (2) The compiler synthesizes implementations for the decode built-ins. These are implemented as matchers for bit patterns.
- (3) The system creates read ports and write ports for the resources. After that, the algorithm assigns these ports to VIR instructions that access resources.
- (4) The next step lowers logic elements to VIR processes. In this stage, the compiler generates the control and hazard detection units.
- (5) The compiler synthesizes the processor core itself. The primary goal is to allocate resources for pipeline registers and interconnect the control and pipeline components.
- (6) The control flow is eliminated and replaced by conditional instructions and multiplexers.

The artifact generators can take over once the compiler has lowered the abstractions. Since the microarchitecture mainly comprises standard VIR processes after this stage, the mapping to an artifact-specific IR is straightforward. The following section elaborates on these steps in further detail.

4.9.1 Instruction Resolving. In VADL each instruction is defined as a separate entity. However, a processor pipeline has to handle *all* of the defined instructions, thus the system first has to establish a comprehensive view over all instructions. This analysis identifies overlaps and commonalities between instructions to make reuse of components possible. For example, the analysis determines the minimum number of read ports needed.

As a consequence of this, the most crucial step in microarchitecture synthesis is integrating the instructions' semantics into the processor pipeline. Figure 7 illustrates the idea of this step. The synthesis must map the two instruction definitions on the left-hand side to the partially displayed pipeline specification on the right-hand side. In this particular case, the algorithm maps three register read operations to the decode stage of the processor and the two additions to the execute stage. Astute readers may notice that inserting three read operations into the decode stage is unnecessary. Because an instruction cannot simultaneously be an ADDI and a SW instruction, the final decode stage should only contain two register reads. Similarly, the two instruction implementations should share the adder that computes the arithmetic operations.

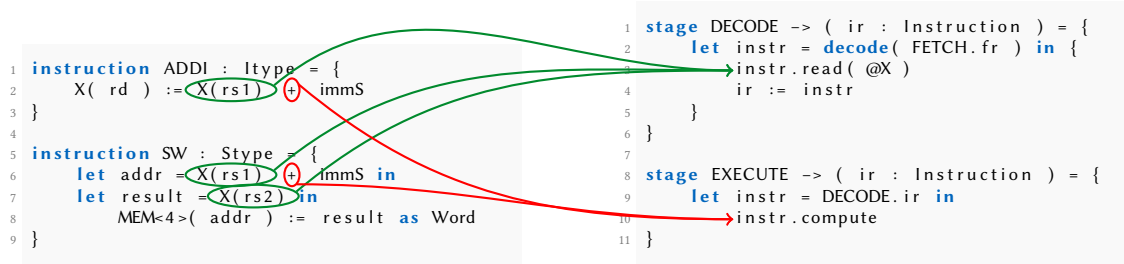


Fig. 7. Exemplary Mapping Between ISA and MiA

The VADL tool tackles all issues mentioned above by leveraging an augmented DFG of all instruction semantics. Figure 8 depicts the DFG for the ADDI instruction from Figure 7. Each node constitutes an operation. Incoming edges denote the input operands of a node, while outgoing edges define how the result of a node is distributed to other operations. The compiler associates each occurrence of an instruction variable in the MiA definition with a DFG. During instruction resolving the DFG is reduced stepwise towards the root nodes, starting from the leaves. Each reduction step results in the emission of one or more VIR instructions and represents the current execution state of the instruction as it progresses through the processor pipeline. Thus, the graph represents the outstanding computations, since all completed computations have already been collapsed in the leaf nodes. A reduction step does not represent a control point in the MiA. Red and blue nodes denote read and write operations, while black nodes denote future pure computations. Green nodes represent values that the processor has already computed. We refer to these values as *available* nodes. For example, the green nodes are the format fields in Figure 8. Figure 9 depicts the DFG for the same instruction after reading the X register file and sign-extending the immediate value.

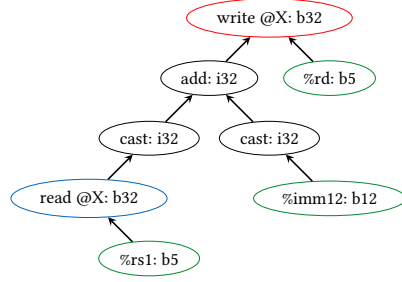


Fig. 8. Simplified DFG for the ADDI instruction

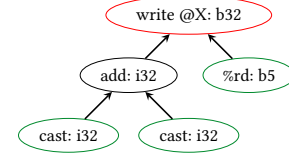


Fig. 9. Simplified DFG for the ADDI instruction after reading the X register file

The real power of this data structure comes from combining the DFGs of all instructions. The origin information is preserved by annotating the nodes with the original instructions. The compiler can then apply global optimizations like coalescing equivalent nodes and reducing the number of read and write nodes. This transformation may require the insertion of nodes that multiplex between values depending on the currently executing instruction. The resulting graph captures the execution progress for the *entire* instruction set architecture. Therefore, this graph is called the Instruction Progress Graph. Figure 10 depicts the IPG for the ADDI and SW instructions from the example from above. The format fields %imm and %imm12 are two different nodes because they are extracted from different parts of the instruction word. Thus the main purpose of the IPG is to synthesize all instructions from the ISA definition and all processor stages from the MiA definition into a single VIR program. Later this VIR program is the input to the Hardware Generator described in 4.11, which in turn generates the actual MiA description in Chisel.

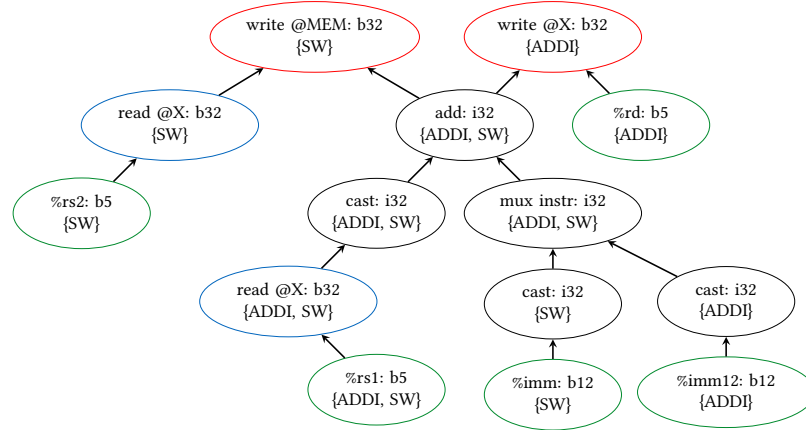


Fig. 10. Simplified IPG for the ADDI and SW Instructions

Now that the compiler has established a holistic view of the execution progress, it can map the IPG to the microarchitecture. This process is called instruction resolving, see algorithm 1. The idea of this approach is to track the flow of the instruction variables across the microarchitecture. If the analysis encounters mappings on these variables (e.g., `instr.read(@X)`), the algorithm replaces the mapping with the actual VIR instructions to implement the instruction semantics. Then, the IPG is updated to reflect the progress. When encountering the next instruction mapping, the

algorithm uses the new IPG to determine the VIR instructions that replace the mapping. The following paragraphs delve into some of the intricacies of this procedure as it is paramount to the microarchitecture synthesis.

Algorithm 1 Instruction Resolving

```

1: stages ← COMPUTESTAGEORDER()                                ▷ Logical order of stages; e.g., decode before execute
2: for all stage in stages do
3:   for all inst in GETRELINSTRUCTIONS(stage) do                ▷ Replace instruction mapping with instruction semantics
4:     var ← GETINSTRUCTIONVARIABLE(inst)
5:     ipg ← GETCURRENTIPG(var, inst)
6:     matching ← MATCHNODES(ipg)
7:     REPLACEINMIA(matching, inst)
8:     newipg ← UPDATEIPG(ipg, matching)
9:     STOREIPG(inst, newipg)
10:  end for
11:  for all param in GETINSTRUCTIONOUTPUTS(stage) do            ▷ Replace instruction abstraction
12:    ipg ← FINDIPGATSTAGEEND(stage, param)
13:    regs ← COMPUTEPIPELINEREGISTERS(ipg)
14:    REPLACEINMIA(param, regs)
15:  end for
16: end for
17: lastipg ← FINDLASTIPG(stages)
18: if lastipg ≠ ∅ then                                          ▷ Check if all semantics were realized in the microarchitecture
19:   ISSUEERROR()
20: end if

```

The algorithm starts by computing a topological order of the stages and their interdependencies. This order ensures that the compiler processes a stage logically preceding another earlier (e.g., decode before execute). The algorithm then iterates over all stages. For each one, the algorithm must complete two necessary steps.

In the first step, the algorithm replaces the instruction mapping with the instruction semantics defined by the IPG. For example, this means replacing `instr.compute` with VIR code that does addition and multiplication. The first problem is to extract the IPG subgraph that matches the current instruction mapping. Each instruction mapping defines a predicate to distinguish between matching and non-matching nodes. This predicate is evaluated for each node, thus partitioning them into matching and non-matching sets. In addition, a node must be *ready* to qualify as a matching node. A node is ready if all its input values are available or become available in this instruction mapping. As a consequence, for example, `instr.compute` cannot match an add node if one of the node's inputs is a register read that is not yet available. After replacing the instruction mapping, the algorithm updates the IPG to reflect the progress. One of the major concerns during this update is marking the computed nodes as available. This procedure can make some nodes unnecessary as they are only inputs to other available nodes. In other words, all computations that require them as input have already been computed. Because the compiler must only keep relevant available nodes in the graph, this step also includes a clean-up process that removes unnecessary available nodes. The updated graph is then associated with the instruction mapping. This link is necessary to ensure the algorithm can access the correct IPG for the next instruction mapping.

The second step in processing a stage is to replace the instruction stage output with the pipeline registers associated with this instruction variable. The available nodes define all computed values necessary for executing the remaining instruction semantics. Recall that the algorithm removed unnecessary available nodes in the previous step. The compiler can quickly determine the VIR instructions that must be saved in the pipeline registers via the maintained mapping.

The compiler can also merge multiple available nodes into a single pipeline register to optimize the usage of resources. This transformation is possible if the nodes are not active in the context of any instruction. For example, a temporary result in an ADD instruction may not be necessary when executing a SW instruction and vice versa. Thus, the algorithm can store both temporary results in a single pipeline register and multiplex between the values. Once the algorithm computes the pipeline registers, it replaces the instruction stage output with the pipeline registers. The mapping from the IPG to the VIR instructions also records the pipeline register of available values. A stage that reads the instruction variable can create probes for these pipeline registers.

There are *no instruction abstractions* in the pipeline definition once the compiler has executed both steps. This fact highlights the importance of this procedure in bridging the gap between the high-level VADL MiA model and a synthesizable microarchitecture. Before concluding, the algorithm ensures that the IPG is trivial, i.e., empty at the logical end of the microarchitecture. If not, the synthesis did not realize some part of the instruction semantics in the microarchitecture. This circumstance is undesirable, and the compiler issues an error to the user. The user can then use the graphical representation of the IPG to debug the issue. This concludes the first step in the microarchitecture synthesis.

While the prototype compiler does not yet support the advanced techniques presented in section 3.4.7, we would like to highlight the importance of the IPG in this context. In the future, the information in the IPG shall be used to automatically determine, for example, the layout of entries in reservation stations. The approach will be similar to computing the set of pipeline registers.

4.9.2 Decoder synthesis. The compiler replaces the decoder built-ins with a bit pattern matching implementation. This is done by analyzing the instruction encodings in the ISA. In the current compiler, the decoder is responsible to deduce the exact instruction (e.g., ADD or SUB) from the instruction word by matching the bit patterns of the opcode. The implementation will compare the constant parts of the instruction encodings with the instruction word. If all constant parts of an encoding match, the algorithm found the corresponding instruction. The comparison order prioritizes tests from more specific encodings. This approach allows for a relatively simple decoder synthesis step. Optimizing away the instruction kind variable is done on the IPG in the instruction resolving step. We rely on the synthesis tools to completely eliminate the instruction kind variable.

4.9.3 Port Inference and Assignment. The next step in synthesizing the microarchitecture is generating and assigning register ports. The set of ports determines the functionality that a register file can provide in a single cycle. For example, a register file with two read ports allows the pipeline to initiate two read requests every clock cycle. The VADL compiler analyzes the stages in the microarchitecture to determine the number of ports that are necessary to enable full parallelization. For example, if a pipeline requires reading two values from a register file in the decode stage and one value in the execute stage, the analysis would assign three read ports to the register. Note that this analysis incorporates information on the control flow. As a result, two mutually exclusive read instructions can share the same read port. Then, the system allocates the generated ports to individual read and write VIR instructions. These processes are isolated as this design separates the generation and assignment procedures. Thus, extending the implementation to allow users to set a maximum number of read and write ports to limit resource usage is straightforward. In contrast to registers, for the main memory our prototype currently only supports a single read and write port.

4.9.4 Logic Element Synthesis. The system synthesizes the logic elements after determining the read and write ports. Each type of logic element has a different synthesis procedure. We will illustrate the idea in the example of the

combinational hazard detection unit. The logic of the synthesized component is based on [76]. This element is responsible for detecting hazards in the pipeline that appear due to the interleaved execution of multiple instructions. For example, an instruction in the decode stage may depend on the value of an instruction in the execute stage. However, the latter writes its result only in a later stage. One solution to this problem is letting the instruction in the decode wait until the result is available. However, doing this requires detecting the presence of a hazard. Observing these circumstances is the job of the hazard detection unit.

Synthesizing this unit requires two types of information. Firstly, the procedure must have access to the entire pipeline to look for operations that may cause a hazard. Secondly, it is necessary to know which parts of the system require and provide values from or to the logic element. The former information can be provided easily by sharing the VIR with the synthesis algorithm. The VIR implements the second piece of information with special logic element operation instructions that can be contained in any other VIR definition. These operations include inputs and outputs that the compiler must relay from and to the synthesized logic element. In addition, as regular VIR instructions, they also convey the location where this information comes from and is required. For example, one such operation in the hazard detection unit checks whether a stage requires stalling. After synthesizing the component, the algorithm usually translates these logic operations into inputs and outputs on a stage or logic element, including any required probing instructions. After the logic element synthesis, the VIR instructions explicitly capture the entire logic of the processor.

4.9.5 Pipeline Synthesis. The next step is synthesizing the pipeline implementation. This step is relatively straightforward. Components are represented as VIR processes. As shown in section 4.3, VIR processes contain probe instructions to access input values. A probe instruction represents taking the value of input lines. The probe instructions of components directly refer to the output parameters in other elements. These instructions indicate whether the result shall be observed in the same machine cycle (e.g., the output of the hazard detection) or the next machine cycle (e.g., computed value in the execute stage). This step first determines which values must be relayed from one component to another. For every connection, the synthesis assigns a wire (same-cycle) or a register (cross-cycle) to the value. This step thus realizes some inputs and outputs as pipeline registers.

4.9.6 Controlflow Elimination. To conclude the microarchitecture synthesis, the compiler tries to eliminate all control flow in the stages and synthesized logic elements. The presence of control flow in a VIR process is represented by it containing more than one basic block at this point. This implies that the process cannot be translated to hardware using combinational logic alone, but has to be synthesized into a state machine with one state for the execution of each basic block. This in turn implies that the execution of the VIR process takes more than one clock cycle.

If the VIR process is the description of a pipeline stage, this situation is especially undesirable, because then one machine cycle takes more than one clock cycle. Under the described circumstances the machine cycle increases due to the fact that the whole pipeline can only advance (i.e., each stage hands down results to the next stage) when the slowest pipeline stage is finished. It also causes the faster pipeline stages to be idle while waiting.

To remedy this situation, control flow elimination attempts to remove as much control flow as possible in order to reduce the complexity of the state machines. In the optimal case, all control flow is eliminated and no state machine is necessary. In this case combinational logic is sufficient to implement the VIR process. Ideally, the algorithm can reduce each component to a single basic block. The VADL compiler can still synthesize a functionally correct microarchitecture if this is impossible. However, each machine cycle is distributed across multiple clock cycles, drastically deteriorating performance. One example of such a problematic microarchitecture could contain two instructions that access the same

port in a single stage. This construct often happens in single-stage implementations as loading the instruction and executing a load instruction currently occupies the same port. This step concludes the microarchitecture synthesis.

Even though we have already invested tremendous efforts into implementing microarchitecture synthesis, more is needed to improve the usefulness of the emitted hardware in real-world scenarios. For example, the memory is currently idealized, meaning read results can be served in the same cycle as they are requested. Another example is the need for a floating point implementation and supporting advanced pipeline techniques, such as out-of-order execution. However, we see no reason why it should not be possible to add these extensions to the current prototype. Furthermore, they do not impact the primary goal of the prototype: demonstrating the feasibility of mapping the ISA to the MiA.

4.10 Cycle Accurate Simulator Generator

Unlike an ISS, a cycle-accurate simulator's (CAS) aim is to model properties of the CPU's microarchitecture such as pipeline stalls and latencies of memory accesses. As mentioned in 4.11, the VADL compiler is capable of generating Verilog via Chisel, a HDL, which can be used for simulation by Verilator. However, HDL generation can take considerably longer compared to generating a CAS, see 5.10. Using a CAS allows simulating microarchitectural aspects while also having the benefit of shorter test cycles for changes in the CPU design. This facilitates analysis of the changes and estimation of their impact on real hardware.

Nonetheless, there are many similarities between HDL generation and CAS generation in order to reuse many generation steps and program transformations. This allows the behavior and properties of interest of the resulting CAS to be as close as possible to the actual hardware design. However, instead of outputting Chisel, the VADL compiler generates C++ code. Each resource definition, such as registers and memory, corresponds to an individual C++ class with corresponding functionality (e.g., read and write functions for memory access). In addition, each stage of the microarchitecture is implemented as a C++ class containing an eval function which executes the corresponding functionality. Each VIR instruction is translated into an equivalent C++ operation. Since hardware is parallel in its nature, unlike most common programming languages, the resulting C++-code might seem less idiomatic to regular software engineers. For instance, consider a feature that might only be used conditionally. Developers would usually use an if-statement to steer control flow. However, hardware often uses 'enable' signals, so a piece of hardware only activates if this signal is set to 'HIGH'. The generated C++ code resembles this behavior (e.g. by using a bitmask where all bits are set to one/zero depending on whether the result of the piece of hardware it mimics is required).

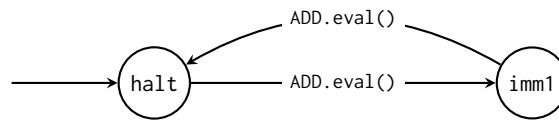


Fig. 11. Example state machine for an ADD instruction which executes the addition and setting the carry flag in two separate machine cycles. halt represents the start and end state. First execution of eval() applies the addition and switches the state to imm1, while the second execution sets the carry flag and switches back to the halt state, finishing the operation.

Furthermore, some operations might require multiple cycles. In order to reflect this, one can partition a single operation into multiple steps represented by states of a finite state machine (FSM). As a toy example, consider an ADD instruction which applies the addition in one cycle and sets the carry flag in the next one. Thus, the instruction consists of two states as seen in Figure 11: A halt state and an intermediate state, which we call imm1. The former state denotes

the start and end at the same time while the latter represents the situation after the addition but before the carry flag has been set. A transition from one state to another requires exactly one machine cycle.

Regarding the generated C++ code, the VADL compiler generates an enum containing an entry for each state. The `eval`-function applies the semantics of the instruction. In the prologue, the method checks what state the instruction is currently in. Considering our example from above, if the current state is `halt`, then the operation is in its starting state. The current state will be updated to `imm1` and `eval` will apply the addition but return before the carry flag will be set. In the next invocation of `eval` (usually after one machine cycle), the method observes that the current state is `imm1` and thus set the carry flag according to the addition which was calculated in the previous cycle. The current state will be updated to `halt`, showing that the operation has concluded.

Last but not least, other components can check whether an operation has finished by querying the `busy` method which is also generated for every stage class. Recall that an operation is busy until it has reached the `halt` state.

The original VADL CAS implementation is not performance optimized. It is very close to the hardware generator to reduce development efforts and to have highest conformance with the hardware. A drawback is the low performance. OpenVADL's CAS will be based on QEMU and try to compute some state statically at dynamic translation time to improve the performance. It is too early to have details of OpenVADL's CAS.

4.11 Hardware Generator

The hardware generator's responsibility is emitting an equivalent hardware design based on a specification in VIR. The microarchitecture synthesis covered in Section 4.9 must have processed the design beforehand. The main objective of this component is to bridge the gap between the VIR and HDLs. For example, in the VIR, ports are global entities that read and write instructions can access. However, in an HDL, the interface of a circuit module must make these connections explicit. The generator uses an HDL-IR to model the hardware design internally. Most IR elements map directly to a concept in a HDL. Thus, emitting files from the IR is straightforward.

This paragraph briefly introduces HDLs for readers unfamiliar with the topic. HDLs enable engineers to model and simulate the behavior of electronic systems before physically implementing them. At the core of HDLs is the concept of a "module." A module is a self-contained unit of hardware description that encapsulates a specific functionality or component of a digital system. Modules can be interconnected to create complex systems and HDLs provide a structured way to define the interactions and relationships between these modules. Users can use logical and arithmetic operations to describe a module's data flow. Furthermore, these languages support concise descriptions of standard hardware constructs like multiplexers and decoders.

Generating the HDL-IR from the VIR is done in three steps. Firstly, the transformation creates a module for each VIR resource definition (e.g., registers). The information on the number of read and write ports is used to generate the module's interfaces. While the process is straightforward, adhering to all constraints in the resource definition is vital. For example, the module must correctly implement hardwired register indices.

After synthesizing the resource definitions, an analysis computes the hardware design hierarchy. The analysis can obtain this information from the instantiation relations between processes. The root of this hierarchy is the behavior process of the Central Processing Unit (CPU). The algorithm recursively enumerates the reachable process definitions and defines a corresponding module. These components contain connectors for all used read and write ports. A parent module must provide dedicated connections for each port used by its child modules. It is mandatory to define these connections explicitly. The transformation combines multiple child connectors to the same port to a single one. This

can be done because the microarchitecture synthesis guarantees that there are no instances where two child processes can access a port simultaneously.

The final step in computing the HDL-IR is generating the actual circuits. This task boils down to synthesizing the modules of nested processes rooted in the CPU behavior. Each synthesized process instantiation requires a description of the module hierarchy, data flow, and control logic. Computing the module hierarchy and data flow is straightforward.

The derivation of the hierarchy naturally unfolds during the enumeration of nested processes. Each process module contains sub-modules that correspond to the instantiated sub-processes. None of these modules includes components for resource definitions. The task of defining the data flow hinges on the premise that most VIR instructions coincide with an operator in the HDL-IR. There is a peculiarity when translating instructions that require access to resources. In such cases, the algorithm must automatically connect the inputs and outputs to the corresponding port interfaces. The HDL-IR does not contain modules for commonly found components such as an Arithmetic Logic Unit (ALU). All computations are done directly within the pipeline stages. Synthesizing the implementation of operations like addition and multiplication is left to the hardware synthesis tool. This approach contrasts with tools that compose the microarchitecture from a set of predefined components.

A fully functioning VADL-generated processor design requires control logic that implements the control flow of processes. Unfortunately, this is often undesirable in a pipeline stage as executing control flow can occupy multiple clock cycles. Executing a basic block requires at least one clock cycle. Therefore, the whole pipeline halts once a pipeline stage executes control flow. The microarchitectural synthesis tries to eliminate the control flow in the pipeline stages to address this issue. However, control flow may still be necessary for the processor's initialization logic. Furthermore, given resource limitations, the hardware generator must resort to control flow for some design specification. For example, the microarchitecture synthesis currently creates a single memory read port. If a pipeline stage requires two memory reads (e.g., instruction fetch and memory operation), control flow is necessary to distribute these accesses across multiple clock cycles.

The generator implements control flow by translating the process to a finite state machine. Each basic block corresponds to a state. The terminator of the basic block determines the state transitions. For example, an unconditional jump will result in a single transition between the two states. Each state machine has an enable and busy signal in addition to its regular inputs and outputs. Initially, the state machine remains idle until the parent module drives the enable signal. The state machine asserts the busy signal until its finished execution. Thus, once the busy signal is low, a parent module is aware that a process has terminated, and the output is valid. The hardware computes all operations in a state simultaneously. Therefore, the implementation must handle side effects with great care once child FSMs run for multiple cycles.

After generating the hardware design hierarchy, the generator emits the corresponding Chisel files. The Chisel compiler then translates the specification into Verilog, a well-established hardware description language. After that, users can apply off-the-shelf hardware synthesis and simulation tools to the generated design. This also allows embedding VADL-generated cores into a larger Verilog (or Chisel) design.

5 EVALUATION

5.1 ISA Language Evaluation

To evaluate the expressive power of VADL we specified different processor architectures with different characteristics. Some statistics about the specification for the instruction set architecture section are shown in Table 1. "Lines of Code"

gives the number of lines of code of the unprocessed VADL specification. "Lines without Comments" are the lines of code which are not comments. "Models" gives the number of model definitions in the VADL specification. "Expanded Lines of Code" is the number of lines of code after model expansion in a pretty printed compact VADL specification without comments. "Function Definitions" and "Format Definitions" give the number of function respectively format definitions. "Instruction Definitions" gives the final number of instructions in the expanded VADL specification.

Table 1. ISA Specification Statistics

	RV32I	MIPS IV	TriLen	AArch64	AArch32	TIC64x	Hexagon	NEON
Lines of Code	161	1131	521	2334	1273	925	1634	2968
Lines without Comments	161	1023	408	2157	1172	877	1385	1844
Models	8	64	21	142	90	47	33	-
Expanded Lines of Code	339	1432	1301	10227	110369	116768	3877	1228
Function Definitions	-	7	2	60	3	9	32	5
Format Definitions	6	8	9	33	11	10	53	22
Instruction Definitions	37	106	123	799	8865	9778	240	140

The first is the RV32I instruction set of the RISC-V architecture as specified in Listing 34 in the appendix. RV32I is a simple 32 bit Reduced Instruction Set Computer (RISC) architecture without multiplication and division and therefore has the smallest specification. MIPS IV is also a simple 64 bit RISC architecture but the specification is more complete. It has a richer instruction set and the specification includes exception handling and system registers. TriLen is a variable length 32 bit RISC toy architecture where immediate values are 8 bit, 16 bit or 32 bit wide. The ISA includes all instructions which are necessary to generate an efficient compiler. The instruction length is either 16 bit, 32 bit or 48 bit. It serves as a testbed for variable length architectures until we have a VADL specification of a reasonable subset of the AMD64 instruction set.

AArch32 and AArch64 are specifications of the complete integer instruction set of ARM's 32 bit and 64 bit architectures. Both ISAs use a status register and have instructions in many variants because of a large set of addressing modes, scaled operands and in particular in the case of AArch32 predicated execution. The specifications make heavy use of higher-order macros. The expansion factor for the lines of code is about 5 for AArch64 and 95 for AArch32. The AArch64 specification benefitted a lot from aliasing of register files with different constraints. This feature reduced the number of instruction variants by about 500 instructions. The high number of function definitions is the result of the very complex computation of immediate values for logic operations. The encoding functions for these immediate values in the LLVM compiler are C++ code using loops and multiple destructive assignments. The same encoding functions written in VADL are specified in a pure single assignment style using functions by employing the technique of divide and conquer. The high number of format definitions is because of the many different instruction formats and format definitions for quite a few system registers.

TIC64x is a VLIW architecture from Texas Instruments. It is used to show the specification capabilities for VLIW architectures with partitioned register files, complex addressing modes, delayed load and branch instructions and predicated instructions. Because of these features and especially predicated execution there is the high expansion factor of 133 in lines of code. In one line of the VADL specification 12 instructions are specified.

Hexagon is another VLIW architecture by Qualcomm. Like TIC64x, it has predicated instructions and many addressing modes, but only a single register file and no exposed instruction latencies. It also features multiple concepts which are currently not covered by the VADL language (such as hardware loops and forwarding results instructions within

the same bundle). Only a small subset of all instructions and addressing modes are currently implemented as a VADL specification.

Both TIC64x and NEON are the testbed for VADL’s tensor definitions. NEON is the Single Instruction Multiple Data (SIMD) extension of the AArch32 architecture. The specification has been developed before the model support was available in VADL. A rewrite of the NEON specification using model will reduce the size of the specification significantly.

All the architectures specified in VADL until now have demonstrated the great capabilities of VADL to develop concise and comprehensive specifications, not only for complex RISC architectures but also for VLIW and SIMD architectures. The main language elements of VADL which contribute to the expressive power in the ISA section are the syntactic macro system, type inference, two distinct ways of format specifications, format access functions, encoding definitions with constraints, enumeration and match, pure functions, register file alias with constraints, specification of VLIW instruction grouping by regular expressions with constraints and finally the tensor definitions using forall.

5.2 MiA Language Evaluation

To evaluate the expressiveness of the MiA, we specified multiple microarchitectures for the RV32I instruction set. Note that these implementations can be easily retargeted to other architectures by, e.g., defining corresponding operations. Table 2 contains the lines of code per specification, the length of the longest pipeline and the number of functional units. The p1 microarchitecture only has a single stage that executes one instruction per machine cycle². The p2 microarchitecture separates the fetching from the decoding and execution steps. The latter two steps are again separated in the p3 microarchitecture. The p5 microarchitecture implements the well-known 5-stage RISC pipeline, while p5_fw adds forwarding logic to the 5-stage implementation. p5_alt specifies the same microarchitecture as p5 but uses the alternative pipeline construct. This concept can define a whole pipeline in a single definition in which stages are separated with a keyword. OoO is an experimental specification of a superscalar out-of-order processor which was introduced in Section 3.4.7. The OoO_max implementation widens the decoders, adds additional functional units, and expands the buffers of the OoO implementation. Lastly, the OoO_cfe further extends the OoO_max specifications. It adds a complex frontend that uses two branch predictors (a slower prediction that may override the quick prediction), thus spending four cycles on total with fetching and decoding the instructions. The results show that the language is able to concisely specify a range of microarchitectures. Identical or similar specifications can be created for the other architectures presented in this section.

Table 2. Microarchitecture Specification Statistics. The * symbol marks experimental specifications that have not been tested for feasibility with the current VADL generators.

	p1	p2	p3	p5	p5_alt	p5_fw	OoO*	OoO_max*	OoO_cfe*
Lines of Code	19	23	31	52	43	58	81	105	133
Pipeline Length	1	2	3	5	5	5	6	6	9
Functional Units	1	1	1	1	1	1	3	6	6

²As mentioned in Section 4.9.6, this may be implemented in multiple clock cycles.

5.3 Evaluation Infrastructure

All performance evaluations were executed on our continuous integration server. The CPU of the machine is an *Intel(R) Xeon(R) W-1370P* running at 3.60GHz, featuring 16 cores and 128 GiB of memory. However the number of cores was irrelevant, since we only measure single threaded programs.

5.4 Evaluation of the Original Generated Compiler

As the *VADL setup*, we used the original LCB to generate a LLVM compiler target based on the RISC-V 32-bit RV32IM VADL instruction set specification, the same as used in Section 5.7. This generated target can be selected by specifying `-target=rv32im` to the `clang` frontend.

As *upstream setup* for comparison serving as baseline, we used LLVM’s standard upstream RISC-V target `-target=riscv32` with the ILP32 ABI. Both compilers are based on LLVM version 17.0.6. And both setups use the LLVM upstream assembler and GNU linker to generate the executable binaries.

As workload we used the *Embench* [4] benchmark suite. In the evaluation of the original LCB, for both the VADL setup and the upstream setup, we used VADL generated CAS RV32-P3 as execution environment. The metric for the comparison was the number of executed machine cycles as reported by the CAS.

Limitations. As LCB is still a work in progress, there exist some limitations in the VADL setup. At the moment the VADL setup could successfully compile, link and execute 18 of 22 benchmarks found in Embench.

- It failed to compile one benchmark, because the instruction selection did not support some needed patterns.
- Of the 21 benchmarks that were successfully compiled, 3 could not be linked against the runtime library. At the moment the VADL setup does not yet support a complete `libc` runtime library.
- So 18 benchmarks could be compiled, linked and executed without error.
- The original VADL setup is not yet capable of utilizing optimization levels higher than `-O0`.

Figure 12 shows the number of machine cycles for the VADL setup relative to the upstream setup which is used as baseline. For some benchmarks LCB gives better results, for some benchmarks the upstream compiler gives better results, on average both compilers have identical performance. We still have to analyze the cause for the differences in performance of single benchmarks.

5.5 Evaluation of the OpenVADL Generated Compiler

The original LCB could not handle long jumps because the necessary LLVM target hooks were not implemented. The OpenVADL LCB supports long jumps, which enables optimizations that were not possible in the original LCB. Additionally, the constant materialization to support the generation of constants which do not fit into a machine instruction’s operand has been refactored. This also resulted in significant improvements for the frame lowering. The labeling of machine instructions introduced in OpenVADL makes it possible to construct a sequence of instructions which can create large constants with fewer machine instructions. The original VADL used a sequence of `ADDI` instructions to materialize a constant where each `ADDI` is limited from `-2048` to `2047`. Therefore, the OpenVADL LCB outperforms the original VADL LCB when large arrays are allocated on the stack. Instead, the OpenVADL LCB can create a `LUI` and `ADDI` pair which reduces drastically the number of instructions to execute. The benchmarks do not allocate large stacks which is why this optimization is not particularly visible in Figure 12.

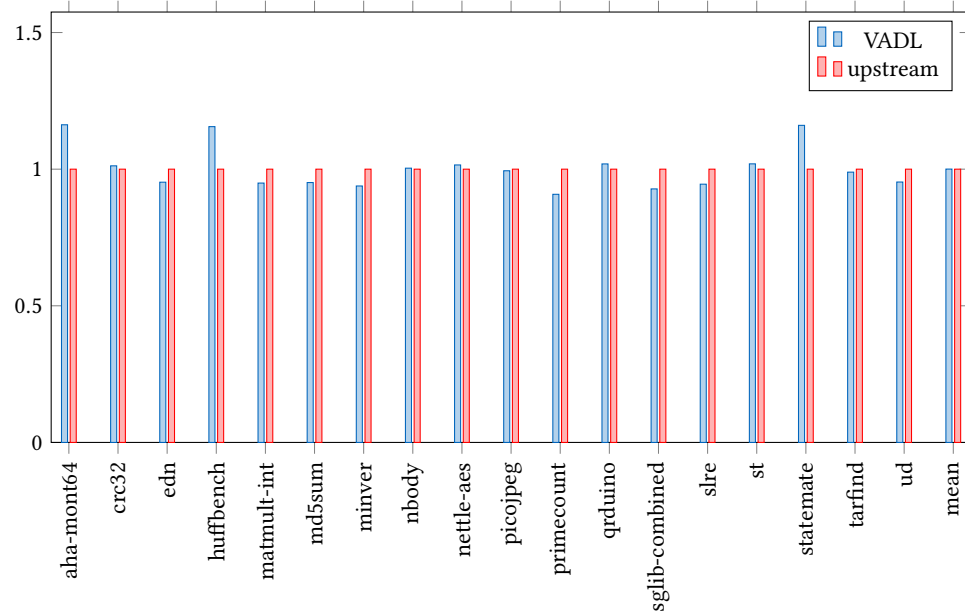


Fig. 12. Embench count of machine cycles of VADL setup, scaled relative to upstream setup (lower is better)

As a performance metric, we used the number of executed instructions reported by the functional simulator Spike [5]. This was necessary, since OpenVADL does not yet provide a CAS, and the original CAS was not compatible with the OpenVADL LCB. Therefore, a direct comparison of Figure 12 and 13 is not possible. All benchmarks have been compiled with `-O0`.

The Figure 13 shows that the mean of the number of executed instructions over all benchmarks is around 6% higher than upstream. LLVM’s upstream implementation emits machine instructions in multiple handwritten C++ classes which are not possible to define in TableGen. OpenVADL’s LCB only produces TableGen patterns, and does not provide these handwritten optimizations.

- Like with the original LCB, it failed to compile one benchmark, because of some missing patterns.
- Of the 21 benchmarks that were successfully compiled, one could not be linked against the runtime library.
- So 20 benchmarks could be compiled, linked and executed without error.

5.6 Evaluation of the Assembler and Linker Generator

The evaluation of the generated assembler and generated linker focused on ensuring that the tools work correctly. We gathered evidence for this by including them in our testing environment. The test infrastructure uses the assembler and linker to build executables for all test programs generated for the RISC-V architecture. These tests include a RISC-V compliance suite, handwritten assembly programs, and the compiler output for small C-Programs. A subsequent step uses these executables to test the generated simulators. The setup exercises the assembly printer (compiler output), assembly parser, machine code emitter, and linker. If one of these components is erroneous, the subsequent simulator tests will fail if the simulator cannot execute the program correctly or the simulation returns a wrong result.

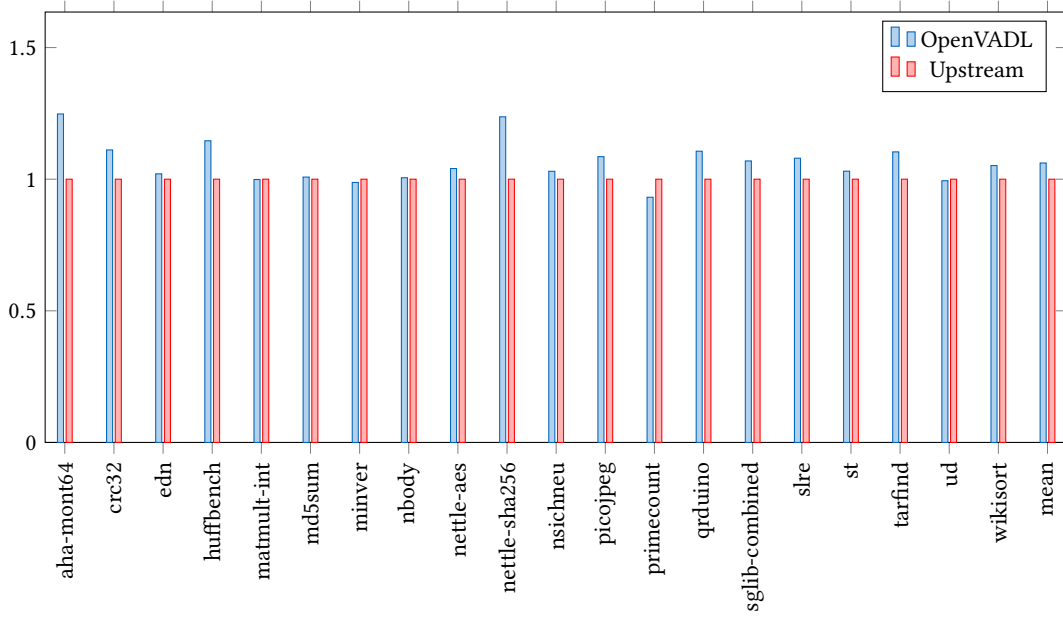


Fig. 13. OpenVADL LCB RV32IM: Embench Spike executed instructions (lower better)

Table 3. Number of Generated Grammar Rules

Name	# Insts	# Inverted	Percent	# Rules
RV32I	74	74	100	75
MIPS IV	106	104	98.11	105
Aarch64	1439	1437	99.86	1446

During development, the VADL team used the RISC-V architecture to test the assembler generator and linker generator. The resulting prototype was applied to the MIPS IV and AArch64 architectures to test the ability to capture different assembly syntaxes. The following text lays out the findings in a qualitative discussion.

Describing the canonical MIPS IV syntax posed no problems. However, abbreviated instructions that leave out some operands lead to problems due to the prototype’s LL(1) parsing algorithm. Similar limitation can also be observed for abbreviated RISC-V instructions. One could extend the parsing algorithm to LL(k) or incorporate a state-of-the-art parser generator to alleviate this problem. Please note that this is a limitation of the prototype and not VADL itself.

In addition, the test with the AArch64 instruction set showed some limitations of VADL’s grammar definition. The problem is rooted in the separation of the parsing and matching phases, i.e., casting a grammar element to an @instruction. As a result, the instruction matcher is oblivious to the applied grammar rule. This is not a problem if the parser conveys the necessary information to distinguish between instructions with equivalent operands to the matcher, e.g., via the instruction’s mnemonic. However, the current prototype only supports communicating this syntax information for mnemonics. As a result, distinguishing instructions based on their syntactical structure proved problematic. Providing additional data to the matcher can remedy this issue.

The proposed grammar rule inference was evaluated on the VADL specifications of the RISC-V, MIPS IV, and AArch64 architectures. The focus was on identifying what type of formatting functions the approach could handle. Table 3 shows the number of instructions in an architecture and the proportion of successfully inferred grammar rules. The last column lists the number of generated rules. This number also includes generated helper rules from the interpretation-based inference mechanism. The system could process all instructions in the RV32I ISA and most instructions in the MIPS IV and AArch64 architectures. The two instructions that could not be inverted in the AArch64 architecture were due to issues in the interpretation of the VIR. Therefore, the issue lies in the implementation of the interpreter and not the grammar inference.

In the MIPS IV architecture, the approach could not deal with the abbreviated syntax of the `syscall` and `ebreak` instructions. This abbreviation emits the 20-bit long code field only if it is unequal to zero. Because conditionals require switching to the interpreter, this would result in 2^{20} combinations requiring evaluation. Section 4.6.5 details this behavior. Fortunately, VADL provides the escape hatch to manually define grammar rules for problematic instructions to overcome this issue quickly. Still, the experiments show room for improvement when dealing with abbreviated syntax.

5.7 Evaluation of the Instruction Set Simulator

We evaluated the VADL simulators using the Embench [4] benchmark suite. For the ISS, we implemented the RISC-V 32-bit instruction set including the M extension (RV32IM) and the complete integer subset of the AArch64 instruction set (Armv8-A). The RISC-V benchmarks were compiled with GCC 12.2.0 and glibc, the AArch64 benchmarks using Clang 14.0.6 and musl libc v1.2.4.

VADL currently has incomplete support for floating point numbers, so the floating point operations in Embench were compiled to instead use software emulation for both architectures (which is normal for RV32IM, but the Armv8-A specification technically mandates hardware floating point support).

The ISSs were compared against QEMU, which by default uses JIT compilation for increased performance. Therefore, we also include QEMU with the `one-insn-per-tb` flag enabled (which JITs every instruction into its own block, negating much of the JIT speedup), and QEMU compiled to use the fallback interpreter (`--enable-tcg-interpreter`) instead of the JIT. These are called "QEMU singlestep" and "QEMU nojit" in the figure below, respectively. All QEMU runs were performed using user mode emulation. For RISC-V, we also included the Spike reference simulator [5].

Figure 14 compares the benchmark runtimes (and their geometric mean) of the described simulators (relative to QEMU). The VADL ISS is more than 21 times slower than QEMU in JIT mode and 75% slower than Spike, but still faster than the other QEMU versions. This is expected because JIT simulation is faster than interpreting while the interpreter Spike is written and optimized for just a single architecture.

As shown in Figure 15, the performance characteristics of the AArch64 ISS are similar to the RISC-V performance characteristics.

5.7.1 OpenVADL ISS Evaluation. Since the OpenVADL ISS is based on QEMU, its primary goal is to achieve performance comparable to handwritten QEMU guest implementations. The comparison was limited to these two implementations, as previous evaluations showed that QEMU outperforms both the original DTC ISS and Spike. As the OpenVADL ISS generator is still in its early stages, the RISC-V specification is currently the only one fully compatible with the supported language features. Therefore, we used the RV64IM ISA specification to evaluate the performance differences between the OpenVADL ISS and the official handwritten QEMU RISC-V guest.

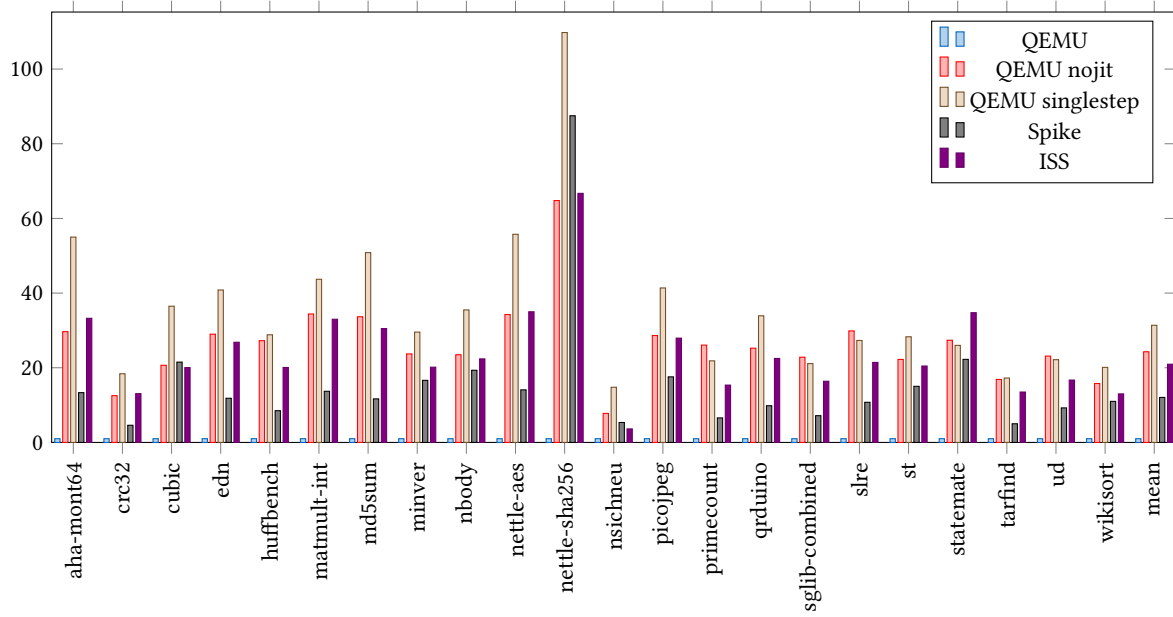


Fig. 14. Embench runtime of instruction-level simulators using RV32IM (relative to QEMU, smaller is better)

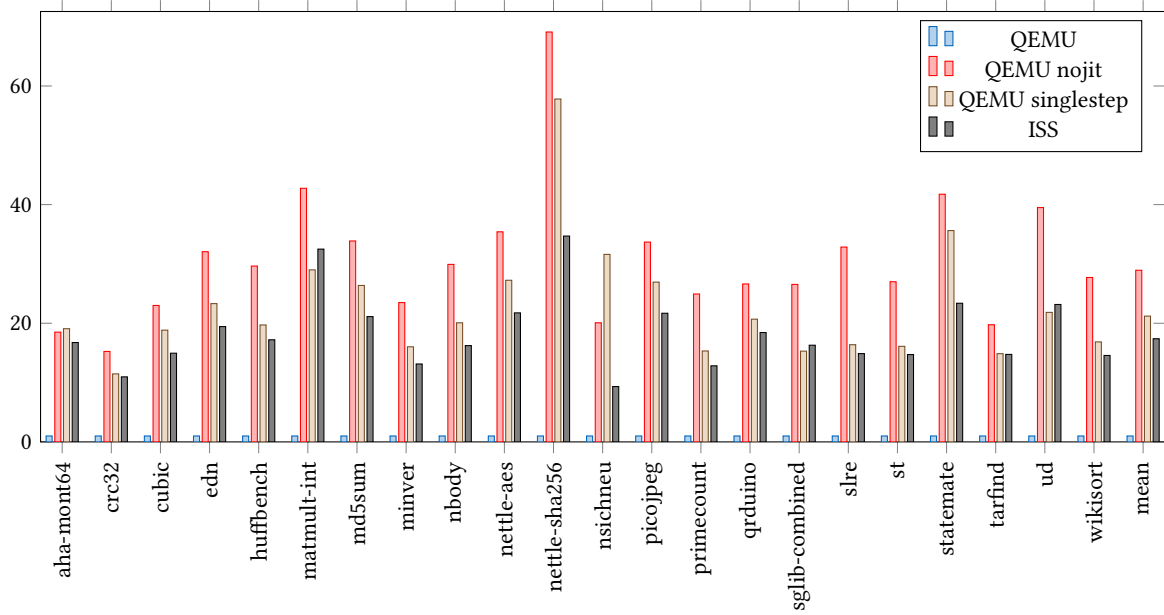


Fig. 15. Embench Runtime of instruction-level simulators using AAarch64 soft-fp (relative to QEMU, smaller is better)

Figure 17 shows that the OpenVADL ISS outperforms the official QEMU implementation in most benchmarks, achieving up to a 40% reduction in runtime. However, there are cases where QEMU is up to 20% faster, likely due to a

missing TCG optimization in OpenVADL that introduces unnecessary move and comparison operations when handling jumps within an instruction. Overall, the generated TCG operations closely resemble those emitted by the handwritten RISC-V guest.

We believe the handwritten RISC-V guest performs worse in some cases due to its overall complexity. It includes various additional features, such as support for the 16-bit compressed instruction extension, which introduces overhead when executing simple programs compiled for RV64IM that do not utilize these features. In contrast, the generated ISS strictly adheres to the given specification, with all other components, such as the machine definition, kept minimal. In terms of emitted TCG operations, handwritten QEMU frontends can be considered optimal. This evaluation shows the advantage of generated simulators over handwritten ones. If the necessary optimizations are provided and if the specification of an architecture is large, handwritten simulators cannot compete with generated ones. An additional advantage is that TCG operations could be added to QEMU and immediately used by generated simulators whereas this would require a big effort for handwritten simulators.

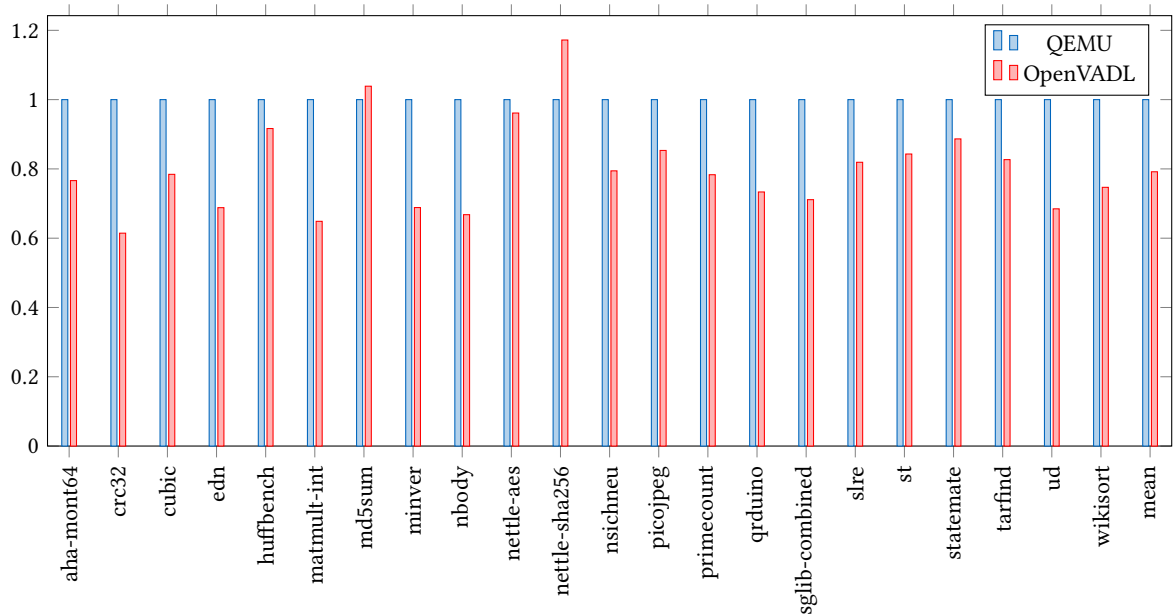


Fig. 16. Embench Runtime of instruction-level simulators using RV64I (relative to QEMU, smaller is better)

5.8 Evaluation of the Cycle Accurate Simulator

We implemented four microarchitectures for the RISC-V 32-bit RV32IM instruction set specification as used in Section 5.7:

- **RV32-P1:** This is a 1-stage pipeline with all steps (fetch, decode, execute, memory, write-back) executing within one cycle.
- **RV32-P2:** This 2-stage pipeline separates fetching and decoding/execution/write-back into two separate stages.
- **RV32-P3:** This 3-stage pipeline has a fetch, decode and execute/write-back stage.

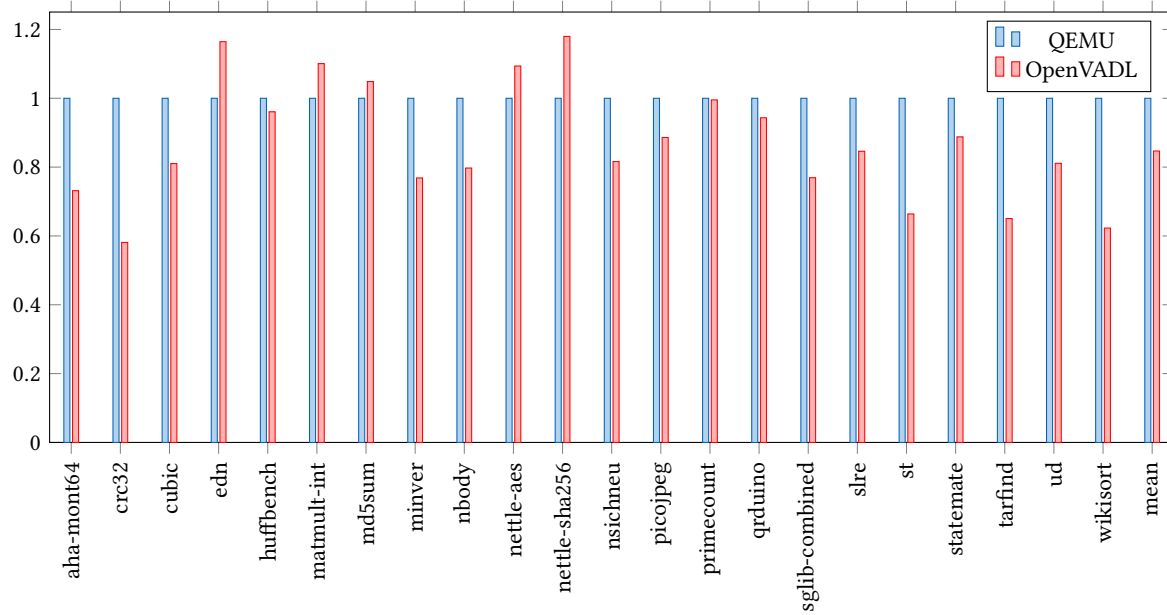


Fig. 17. Embench Runtime of instruction-level simulators using RV64IM (relative to QEMU, smaller is better)

- **RV32-P5:** This 5-stage pipeline has a fetch, decode, execute, memory and write-back stage.

We compare the CASs generated by VADL to gem5 using its AtomicSimpleCPU. Atomic refers to the memory subsystem, meaning that memory accesses return instantly. We chose to use this model as the CAS does not accurately simulate the memory subsystem yet. The CPU simulated by gem5, however, does not have a pipeline, which is unrealistic for real-world CPUs. Thus, we decided to provide a 1-stage pipeline microarchitecture in VADL to allow for a fair comparison.

Figure 18 shows the runtimes of the Embench benchmarks with the three CASs and gem5 (relative to the one-stage CAS). The CAS becomes slower as more stages are added (in this case three stages is more than 5 times slower than a single stage) because there is an overhead associated with simulating a longer pipeline. gem5's performance lies between the 1-stage and 2-stage CAS.

Figure 19 compares the cycle counts of the Embench benchmarks, gem5's (estimated) cycle count aligns with the one-stage CAS. The simulators with two or more stages each have higher cycle counts (in this case three stages have a 62% higher cycle count than a single stage) which is expected because the real processors would then be able to run with a (much) higher clock frequency, making the processor faster overall.

The CAS was created to provide a faster cycle accurate simulation than a low level HDL simulation using Verilator version 5.010. This was evaluated in Figure 20: currently the CAS is slower than the equivalent HDL compiled into a simulator using Verilator. This issue stems from linearizing the stage computations so that each stage must only be executed once by the CAS. However, currently, this leads to duplicate computations in multiple stages. Verilator handles these situations better than the C++ compiler, thus outperforming the CAS. We are working on addressing this issue. As expected, the ISS is three orders of magnitudes faster than verilated HDL or the CAS, with gem5 being slightly faster than the HDL simulation.

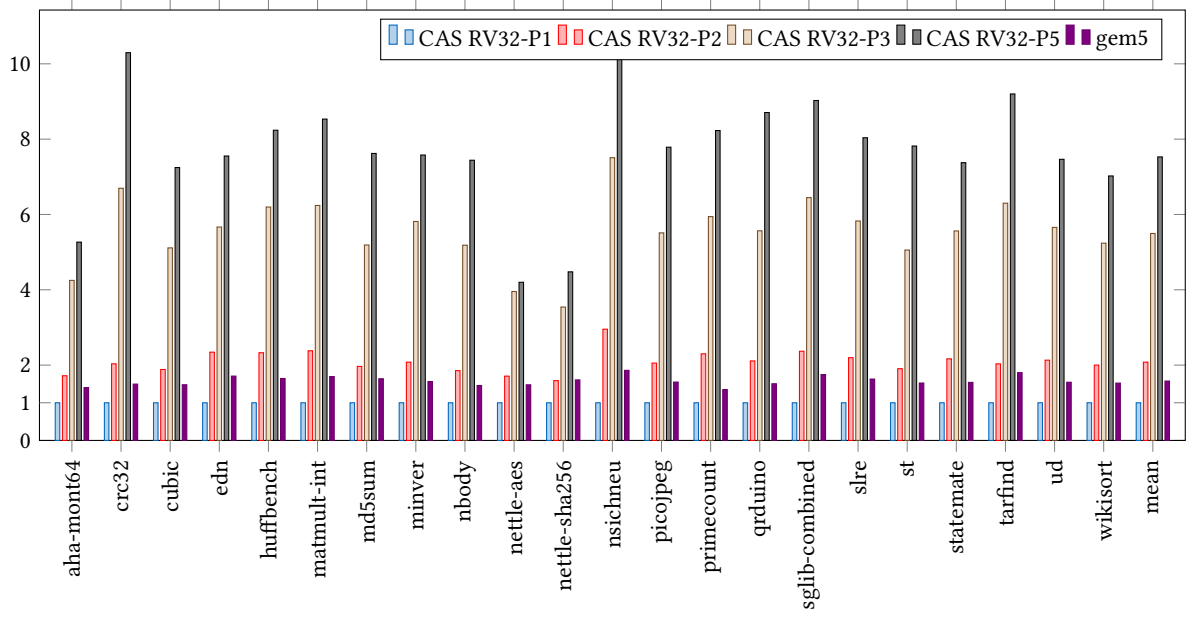


Fig. 18. Embench runtime of cycle-level simulators using RV32IM (relative to CAS RV32-P1, smaller is better)

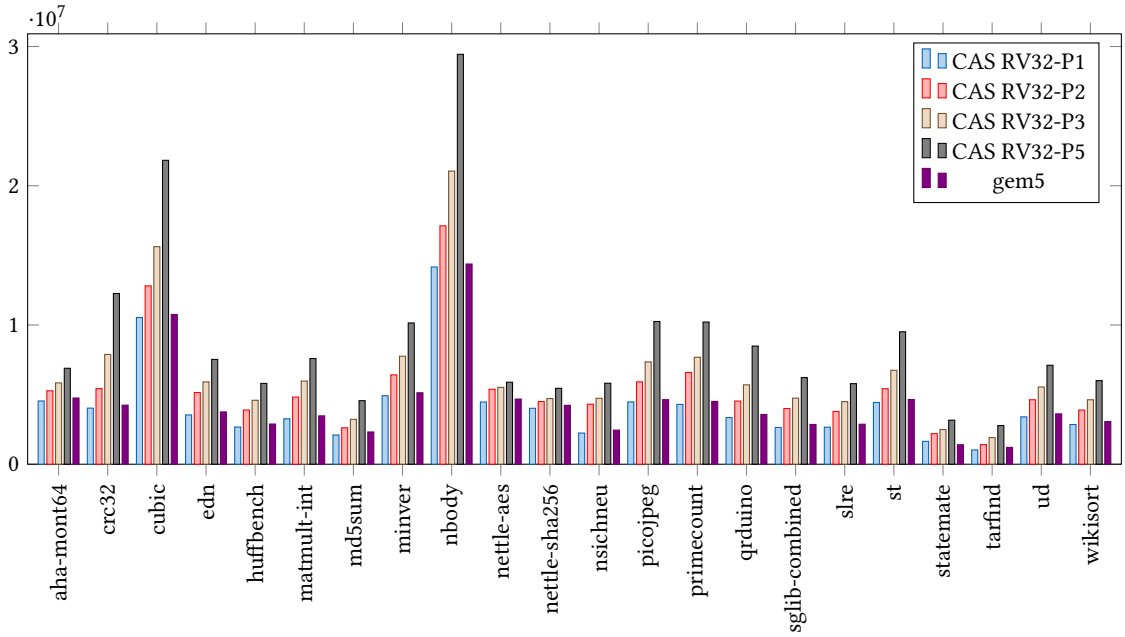


Fig. 19. Embench simulated cycle count using RV32IM

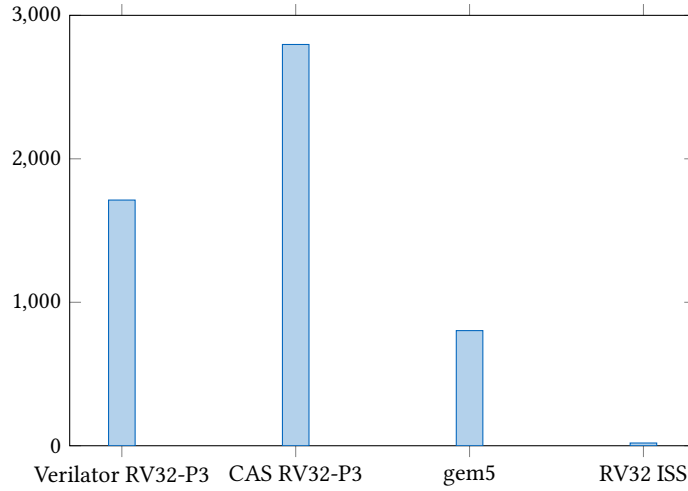


Fig. 20. Embench runtime of RV32IM, comparing with HDL Verilator (in milliseconds, geometric mean over all benchmarks, smaller is better)

5.9 Evaluation of the Hardware

We evaluated the hardware generator similarly to the CAS. The VADL tooling generated Multiple Chisel implementations of an RV32IM-compliant processor. Each implementation was translated to Verilog and simulated using Verilator version 5.010. Then, the test setup validated the designs by running the Embench benchmark suite and comparing the actual output with the expected one. Furthermore, a RISC-V compliance suite was executed on every verilated design. All designs exhibited the desired behavior in all test runs.

Quality is another crucial aspect of the designs, as it impacts the realized processor’s performance, power consumption, and chip area. The generated designs are compared to hand-crafted implementations of the same ISA. The Sodor standalone open source designs (revision e5638c39e5750ea98527547fbc3f9d269c451f3a) will be a reference in this work. Once the VADL tooling is mature enough to handle more sophisticated concepts (e.g., reorder buffers), future work may compare the generated artifacts to industry-grade processors. Before continuing with the evaluation, we want to highlight that both the VADL-generated designs and Sodor used idealized memory. The result of the comparison may be different for real-world memory modules.

We compared the structural metrics of the corresponding 5-stage implementations using the tool Yosys version 0.29. To get a chip area metric for comparison, we used a simple demo cell library and mapping script found in the Yosys distribution. In order to achieve a fair comparison, we chose a VADL specification similar to the 5-stage Sodor. This specification contained forwarding and does not implement the RISC-V M extension for multiplication, which is not present in Sodor.

The RISC-V ISA specifies the possibility of including up to 4096 Control and Status Registers (CSRs) in a RISC-V implementation, each 32 bits wide. The CSRs are used to manage privilege modes and to provide general information about the processor state, e.g.:

- Vendor ID
- Machine trap handling

- Machine memory protection
- Machine counters
- Debugging information
- Custom information, specific to a concrete implementation

Not all CSRs have a standardized usage, leaving the possibility for custom extensions. Since most CSRs are optional, Sodor only implements a subset of CSRs resulting in a much smaller CSR register file than the complete one used in our VADL specification.

Table 4 shows the numbers reported by Yosys’ stat utility. Since the VADL hardware generation does implement the *full* CSR RISC-V register file, which is $2^{12} * 32 = 131072$ bits, we also provide the numbers without the chip area used for the CSR modules.

In the future the VADL hardware generator will be able to restrict the number of implemented registers.

Table 4. Comparison of chip area for VADL RV32I and Sodor

	VADL RV32I	Sodor	factor
Total area	10620711.0	170811.0	62.18
Area without CSR	203168.0	114785.0	1.77

5.10 Evaluation of the Simulator Build Times

For productive experimentation and design space exploration it is beneficial to have short simulator build times. Depending on the task at hand, a short edit-build-run cycle also can influence the decision which kind of simulator to choose.

We measured the generation time of the RISC-V generators used in our evaluation for comparison. Table 5 shows the complete build times of the simulators for the RV32IM specification as reported by bash in seconds. The generation of ISS and CAS include the VADL execution time and the build time of the emitted code. The generation of HDL includes everything from the VADL execution time, the translation from Chisel to Verilog, the execution time of Verilator, to the build time of the code emitted by Verilator.

Table 5. Build times of RISC-V RV32IM simulators

simulator	build time in s
ISS	6.686
CAS 1 stage	10.779
CAS 2 stages	14.357
CAS 3 stages	12.480
CAS 5 stages	12.348
HDL 5 stages	58.599

6 RELATED WORK

PDLs

A PDL is a domain specific Architecture Description Language (ADL) which is used for specifying and designing a processor architecture. Typically a PDL is capable of describing aspects and properties of a processor in a succinct and

convenient way. According to [92] PDLs can be classified regarding their *content*, i.e., what the PDL describes, and regarding their *objective*, i.e., what the specification can be used for.

Classification by *content* distinguishes between *structural*, *behavioral* and *mixed* PDLs. The main focus of a structural PDL is the possibility to describe the hardware components constituting the processor and the interactions between these hardware components, e.g., registers and processor pipeline. A behavioral PDL focuses on describing the semantics of the instruction set supported by the processor. While a structural PDL provides information about the hardware not present in a behavioral description it is in general difficult to infer instruction semantics from a pure structural description of the processor. However, [81] and [24] showed that with some effort it becomes feasible. Most PDLs are mixed, showing characteristics of both structural and behavioral PDLs, but with different emphasis.

Usual objectives of a PDL are *compilation*, *simulation*, *synthesis* and *validation*. A PDL focusing on compilation is used as input to a compiler generator. It must provide accurate information about the semantics of instructions, which a behavioral PDL is well suited for. However, it should also provide structural information about the processor, e.g., information about pipeline stages or functional units, to determine an accurate cost model. This cost model of the processor is used in the generated compiler. With the objective of simulation, it depends on the type of simulator that can be generated from a description. A purely behavioral PDL is sufficient to generate an ISS, but a structural description of the hardware is necessary to generate a CAS and the instruction scheduler for the compiler. For hardware synthesis, a structural PDL is sufficient. Formal verification or automatic generation of test cases can benefit from both a structural and a behavioral description, depending on the verification or the test scenario.

Table 6. Comparison of processor description languages

	language classification	redundancy of specifications	generated simulator	cycle accurate	generated compiler	hardware generation	separate format
VADL	mixed	single	DBT	yes	LLVM	full	yes
xDSPcore ADL	behavioral	multiple	compiled	yes	OCE	decoder	no
xADL	structural	single	DBT	yes	LLVM	full	yes
StoneCutter	behavioral	ISA only	none	Chisel	none	partial	yes
CoreDSL2	behavioral	ISA only	source JIT	no	extensions	extensions	no
Sail	behavioral	ISA only	interpreted	no	none	no	partial
ViDL	mixed	single	interpreted	yes	none	full	no
LISA	mixed	multiple	compiled	yes	CoSy	full	partial
Expression	mixed	single+mappings	interpreted	yes	EXPRESS	full	no
nML	structural	single	interpreted	yes	CHESS	full	no
ISDL	structural	single	interpreted	yes	AVIV	full	yes
TIE	structural	single	RTL	yes	GCC	extensions	no
ISAC	mixed	single	multiple	yes	LLVM	no	partial
CodAL	mixed	single	multiple	yes	LLVM	full	partial
ISADL	behavioral	encoding only	interpreted	no	assembler	no	yes
ArchC	mixed	multiple	interpreted	yes	LLVM	no	yes
MAML	mixed	multiple	mixed	yes	LCC	full	no
MIMOLA	structural	single	events	yes	MSSQ	full	no

VADL is a mixed PDL which comprises a single specification split into the distinct ISA and the MiA descriptions. There are no redundancies at all in the specification. Unique language features are the syntactic macro system and the MiA description at a very high abstraction level with no intermingling of ISA and MiA. VADL enables clean

specifications with type, format, constant and function definitions, enumerations, aliasing of register definitions, and format field access functions. It has been designed to ease the development of generators. Different compilers, ahead-of- and just-in-time, are supported. It assimilates the experience of all the existing languages and adds unique new features. *VADL* was the third PDL developed in our research group. The first was an ADL designed for the development of the *xDSPcore* processor, a compiler-based configurable Digital Signal Processor (DSP) [75].

The *xDSPcore ADL* is based on eXtensible Markup Language (XML). A specification contains information about the architectural state including pipeline stages and the kind of instructions a stage can execute. The behavior of the instructions for the simulator and their patterns for the compiler’s tree pattern matching instruction selector are a kind of macros for the generators. This duplication of the instructions’ behavior descriptions leads to redundancies in the specification. The generation of the optimizing compiler is described in [45], the generation of the compiled instruction set simulator is presented in [44]. The compiler supports autovectorization [102], if-conversion of predicated instructions, DSP addressing modes and hardware loops. An instruction decoder in a HDL can additionally be generated from the instructions’ specification. With the *xDSPcore ADL* we experienced the problems of redundant specifications. Therefore, we designed *VADL* in such a way that redundancy does not happen.

The structural PDL *xADL* was the second language we designed. It is described in detail in the thesis of Florian Brandner [23]. It allows the specification of the structure of a processor on RTL in a language with XML syntax. An extremely fast tiered CAS with three levels of optimization is generated from the low level specification. The simulator starts with interpretation, then translates basic blocks and finally complete regions to the host architecture deploying basic block duplication [25]. Despite the low level specification the compiler generator can extract all necessary information to generate a backend for LLVM [24, 27]. The low level specification requires a rewrite of a processor specification if the microarchitecture changes. The problems mentioned above influenced us to design *VADL* as it is now, with strict separation of ISA and MiA.

StoneCutter is a language to define instructions and the hardware pipeline at a high abstraction level [70, 78, 79]. It is only designed to generate a hardware description in Chisel. While no compiler or simulator can be generated directly, Chisel can generate a simulator of the hardware. It is necessary to explicitly specify the splitting of an instruction’s behavior between pipeline stages which results in verbose and error-prone specifications. The language allows multiple assignments to local and global variables and has C-style loops. *VADL* has a higher level of abstraction and supports compiler and simulator generation.

CoreDSL 2.0 is a very simple ISA description language. A simulator based on C source code strings which are compiled just-in-time is available [71]. There is no way to specify compiler or micro architecture related information. Nevertheless, *CoreDSL* has been used for the specification of ISA extensions for semiautomatic generation of compiler and hardware extensions [96, 119]. In contrast to *VADL*, *CoreDSL* does not support formats, zero registers, subword registers, type definitions, enumerations or specification of atomic instructions. Therefore, *CoreDSL* processor specifications are very verbose and error-prone.

Sail [11] is a recent purely behavioral PDL for describing the semantics of an ISA with a focus on the verification of ISAs. It can be used to produce a less performant ISS but also definitions for proof-assistants to provide evidence of correctness. It is not possible to generate a compiler or hardware from a *Sail* specification. *Sail* has a powerful but expensive type system, liquid types. *VADL* has a much simpler type system which can be efficiently checked. Using *VADL*’s enumerations, safety can be ensured in many cases where *Sail* uses its liquid types. *VADL* does not support the rich verification universe of *Sail*, but can generate efficient simulators, compilers and hardware. *VADL* and *Sail* have different design goals.

ViDL, the Versatile instruction set architecture Description Language, is a language to formally specify the instruction set architecture of processors in a functional style inspired by the programming language Standard ML (SML) [37, 38]. An outstanding feature of *ViDL* is that all micro architecture properties are automatically derived by providing the target frequency of the generated processor. There is no specification of any micro architecture details at all. The *ViDL* generators produce a hardware description in VHDL and interpreting simulators. In the specification of the instructions' behavior concurrent assignments happen to storage elements (memory, registers). No multiple assignments are possible to intermediate variables. `let` expressions, as used in SML or VADL, bind expressions to names and define the scope of the name. The set of primitives available for the specification of the instructions' behavior can be extended by specifying the emitted code for the generators (C++ for the simulator, VHDL for the hardware). The generation of a compiler is not supported. With VADL many micro architecture properties are automatically determined, but the MiA can be specified on a high abstraction level, whereas in *ViDL* the complete MiA is automatically determined.

When *LISA* [65, 101, 109] was initially developed, its focus was on efficient simulation of DSP architectures. It is a mixed PDL and can be applied in the generation of many artifacts and tools, for example, a compiler, linker, profiler, CAS or a low level hardware description for hardware synthesis. The behavior part of the language allows arbitrary C-code in small chunks for every stage of a pipeline. This results in very large specifications and requires the addition of a separate description of the compiler semantics. Therefore, *LISA* is very verbose and error-prone. VADL avoids all these problems by the strict separation of ISA from MiA.

Expression [55, 60, 92] is a PDL with a syntax similar to the Lisp programming language. Its main use case is the development of System-on-Chip (SoC) architectures. *Expression* is used to generate a CAS and a compiler which optimizes for Instruction Level Parallelism (ILP). The toolchain is able to generate synthesizable hardware and has extensive support for verification using model checking. *Expression* is a mixed PDL. The main focus is on a low level structural specification like ports, connections, pipeline or caches. The behavioral aspects are described in operation specifications which include operands, behavior and encoding. Mappings from a sequence of operations to another sequence of operations can be specified to ease compiler generation and optimization. In comparison with *Expression* VADL has a more user friendly syntax and specifies the MiA at a higher abstraction level leading to more concise and maintainable specifications. In the current state of implementation OpenVADL does not have the same extensive verification support as *Expression*.

nML [46, 92] is a structural PDL with an attributed grammar at its core. The grammar describes the processor's instructions. The instructions' semantics are specified with an *action* attribute. Further attributes carry cycle count and stalling information. A *skeleton* specifies structural elements carrying the processor state and transitions between elements. The first language versions did not support pipelining, which was later added to the *action* attribute. From a *nML* description the user can generate a compiler, a CAS, a low level hardware description and a test-program generator. *nML* suffers from the intermingling of ISA and MiA leading to complex specifications which are difficult to maintain, a problem which VADL avoids by the strict separation of ISA and MiA.

The syntax of *ISDL* [57–59] is based on an early version of *nML* which has been extended. *ISDL* is a PDL specializing in the description of VLIW processor architectures. It is a behavioral PDL and emphasizes the description of instruction semantics. Descriptions in *ISDL* can be used to generate an assembler, a compiler or a simulator. In comparison with VADL *ISDL* has the same shortcomings as *nML*.

TIE, the Tensilica Instruction Extension language has been designed to support the comfortable extension of the instruction set of the Xtensa processor [53, 121]. The basic instruction set and architectural state is fixed and cannot be changed. *TIE* is based on a subset of the hardware description language Verilog. Because of that the specification is

at a lower abstraction level in comparison to VADL. A compiler, assembler and linker based on the GNU toolchain is generated. The added instructions can be accessed as intrinsics only. Simulation is done on RTL. VADL is more powerful as it can describe any processor architecture and not only extensions of a single architecture.

The ISAC (Instruction Set Architecture C) language is a mixed PDL which was largely inspired by LISA and developed at the Brno University of Technology [104]. Similar to LISA the behavior is specified in a subset of C. The design and implementation of a fast cycle accurate interpreted simulator based on ISAC and using finite state machines for simulation is presented in [105]. Additionally compiled and JIT compiled simulators have been developed. To avoid the duplication of the behavior semantics for the compiler generation, it is extracted from the instructions' assembly grammar. The details of the automatic generation of a C compiler from an ISAC specification are described in [67]. In VADL there is no need for instruction extraction, as there is an explicit definition for each instruction.

With the experience of the ISAC language *CodAL* has been developed [103] and commercialized by the company Codasip Ltd. [6]. An example of the specification of an instruction for a RISC-V extension is available in [9]. There is no further information publicly available on the details of CodAL.

ISADL is an encoding description language for VLIW architectures [125]. Its main purpose is to specify or automatically generate an encoding from a specification of the requirements of the encoding. Requirements can be the set of functional units, the set of encoding fields, the number of instructions, and the default instructions. An optimizing assembler can be generated from the encoding specification by specifying the syntax and matching by regular expressions. The generation of an instruction set simulator is mentioned, but there is no information available on how the semantic of the instructions' behavior is specified, probably code in some high level language like C++. The language is very verbose and the specifications are not easy to understand. In contrast to VADL, neither a compiler nor hardware can be generated.

ArchC [14] is an extension to *SystemC* [97]. Both languages are embedded as a library in the C++ programming language and thus are embedded Domain Specific Languages (eDSLs). While it is possible to describe a processor in *SystemC*, the description is at such a low level that it is impossible to derive structure, semantics of instructions or assembly language representation. ArchC provides a higher level of abstraction and adds this capability. The behavior for every instruction including all MiA details like program counter updates and distribution to pipeline stages is described in *SystemC* (C++). This intermingling of ISA and MiA leads to huge specifications which are hard to adapt to changes in the MiA. An interpreting and a compiled simulator can be generated from an ArchC specification. If the instructions' behavior descriptions include all MiA details, the simulator is cycle accurate. Additionally, an assembler and linker for the GNU toolchain can be automatically generated. Later compiler generation was added [12]. Because the instruction behavior is not suited for compiler generation, an additional semantic specification in an additional compiler semantic language has to be added. This duplication of the specification will lead to specification errors. VADL does not have any of the above described deficiencies.

The MACHine Markup Language *MAML* has been designed for rapid development of application specific instruction set processors with focus on multiprocessor systems [47, 92]. The syntax is based on XML, but specifications can be created using a graphical integrated development environment. Instruction behaviors are specified in C++, necessitating an additional specification of the patterns for the compiler's instruction selector. A mixed compiled/interpreted simulator is automatically generated. The machine model for the compiler's optimizer and scheduler is automatically extracted from the architecture specification. Synthesizable hardware is generated. VADL has a more readable syntax than MAML and does not need graphical editing. In contrast to MAML, VADL can generate the instruction selector of the compiler automatically and has no redundant specifications.

MIMOLA was the first PDL to be developed [88, 92]. It is a structural PDL with PASCAL-inspired syntax and a focus on hardware synthesis, generating a low-level hardware description. Despite the purely structural processor description it is possible to extract instructions' data flow graphs. These are used in the MSSQ compiler which is described in detail in [81]. A distinguishing feature is the possibility to provide typical workloads with the processor description. These are used for profiling and optimization of the resulting hardware. VADL abstained from a structural specification as with changing micro architectures the whole processor specification has to be rewritten.

Additional languages are described in the book by Prabhat Mishra and Nikil Dutt [92].

Retargetable Compilers

An important artifact generated by VADL is the compiler. It is generated by VADL's LCB.

Principles of compilers are described in [7], including retargetability. [82] describes retargetable compilers in the context of embedded systems. This subsection's organization is in large parts similar to [54]. See Section 2 for a short explanation of retargetable compilers.

An early example of a retargetable compiler is described in [49]. A very well established open source compiler that supports a great number of target architectures is *GCC* [116].

Another frequently used compiler is LLVM [77]. It was designed with retargetability as a central aspect. This is evident by the use of a common and well-defined IR which is suitable for a wide range of transformations and optimizations. LLVM uses the *TableGen* language to succinctly specify target specific properties, further simplifying retargetability. *TableGen* is also used to specify the available instruction for a target architecture. The VADL LCB emits a compiler as an LLVM target. Therefore an important task of LCB is the generation of the relevant *TableGen* patterns for instruction selection.

Simulation

This subsection is organized in large parts similar to [111]. See Section 2 for a short explanation of simulation in the context of VADL. A simulator is particularly useful during the development phase of a processor when actual hardware is not yet available because the architecture is still subject to change as described in [26]. [92] mentions a simulator's usefulness during design space exploration.

There are two main goals when constructing a simulator: Accuracy and performance. Accuracy is a measure of how similarly the simulator behaves to the system it simulates, i.e. how well the metrics reported by the simulator coincide with a real run of the simulated system. Performance is a measure of how many computational resources are necessary to execute the simulation. There exists a trade-off between these two goals as a more accurate simulation in general requires greater computational resources. [126] and [127] investigate these trade-offs.

[51] has shown that a simulator enhanced with a graphical user interface can serve as a valuable educational tool to teach the inner workings of a processor. An educational application is also a planned future use of VADL.

Another important way to improve the performance of a simulator is *caching*. [17] shows how caching of decoded instructions that are executed multiple times can be done in a threaded code interpreter. A similar implementation of this caching scheme developed for the *SimICS* simulator [87] is described in [86]. Also the ISS generated by VADL follows a similar caching scheme. The method described in [107] caches not only decoded instructions but also fetched values to improve simulator performance. Caching techniques found in JIT compilers can also be adapted for simulators, like the one found in [83]. These are not implemented in the VADL ISS, but are possible extensions.

Self-modifying code often hampers caching in a simulator. [72] investigates strategies to handle self-modifying code for ISS. The VADL ISS uses the approach described as *value checking*.

Another aspect to consider in a simulator is the handling of *system calls* to the operating system's kernel. [26] describes two common ways to provide this functionality. One is User Mode Emulation (UME), also known as delegation. It forwards system calls from the simulated program to the host's operating system, as implemented for example in [13]. The other is *full system simulation*, which also simulates operating system functionality inside the simulator. Full system simulation is more complex to implement than UME, as it has to consider more components and properties of the guest system, like input/output, access to devices or the memory model, as described in [120]. [29] shows that this technique achieves a higher degree of precision modeling the behavior of the examined processor. QEMU [19] is capable of both modes. At the moment both the VADL ISS and the OpenVADL ISS support UME, but not full system simulation. Since the OpenVADL ISS targets QEMU, we expect its extension to full system emulation to be straight forward.

Interpretive simulation is the most basic simulation model. [73] compares a classical interpreter with a direct threaded code interpreter [18] and an indirect threaded code interpreter [36]. [41] also compares threading models and [42] adds findings about reducing branch mispredictions. The VADL ISS is a DTC interpreter.

Compiled simulation is described in the article [91]. [16] further improves this approach, especially with respect to code size. [108] shows an extension that is able to detect code changes, thus offering the possibility to handle self-modifying code. [44] describes a CAS based on compiling basic blocks to improve performance. It is also capable of switching between interpreted mode and compiled mode.

DBT was first used in Shade [31] an ISS and profiler. The OpenVADL ISS targets QEMU [19], which is based on DBT. *gem5* is an open source computer simulator that has evolved over the years [22, 85] and was extended to model a processor on the RTL [84]. It was used in the evaluation of VADL.

In order to reduce the computational complexity of a CAS there exist cycle *approximate* approaches like [68] or [48]. These approaches reduce accuracy to gain execution speed. A cycle approximate simulator based on QEMU is planned for OpenVADL.

HDLs

In the context of VADL a HDL is a domain specific language used to specify digital electronic circuits. These circuits can be instruction set processors, as in VADL, but also any other kind of ASIC. Compared to a PDL, a HDL is more general, but cannot use domain specific abstractions because of this generality.

The abstraction level on which HDLs operate is called the RTL. See the background section 2 for a short explanation. The step from the RTL to the concrete logic gates (the netlist), the layout and routing of the physical components and connections that make up the electronic circuit, is realized with synthesis tools. Many of these tools are commercial closed-source products, however Yosys [123] is an open source synthesis suite. It can target both FPGAs and ASICs. The algorithms and basic principles for RTL synthesis are described in [56]. VADL's hardware generator outputs code on RTL level, and can be further processed by such synthesis tools.

The most frequently used HDLs are *VHDL* [2] and *Verilog* [1]. These languages are the industry standard. Yosys uses Verilog as its input. An extension to Verilog is *SystemVerilog* [3], which enhances the language with better verification capabilities, user-friendly syntax features and better object oriented abstractions.

Often HDLs are embedded as specialized libraries in general purpose programming languages. One frequently used example is *SystemC* [97]. It is embedded in C++ and offers object oriented features and also good support for simulation. The HDL used in the VADL project is *Chisel* [15]. It is embedded in the Scala programming language and

naturally supports a functional programming style provided by the host language. Chisel is translated to Verilog. See the background section 2 for a short summary.

7 FUTURE WORK

The original VADL implementation and OpenVADL are still research prototypes. Therefore, there are many different areas to expand the research and to advance OpenVADL to achieve production quality.

The core VADL language design is quite complete. We plan to extend VADL to support heterogeneous multiprocessor systems. This requires the specification of bus protocols. Currently, VADL relies on co-simulation and trace comparison to verify the specification and the generated artifacts. We want to extend the language with further verification capabilities.

In addition to extensions of the language, there are a lot of opportunities to improve the generators. Primarily it is necessary that the generators support the entire specification language. This includes support for floating point.

For the compiler generator, we are currently working on efficient code generation for instructions with multiple results and automatic translation of nested loops to tensor instructions. In order to convincingly demonstrate the flexibility of the GCB it is necessary to implement generators for JIT compilers and the GNU compiler collection and its toolchain.

For the simulator generators, we are currently working on support for VLIW architectures. Further optimizations to reduce the overhead in the current CAS are essential. This will be accomplished by using QEMU as the basis for OpenVADL's CAS and by performing statically determinable computations during JIT compilation.

We are working on accurate cache and memory simulation. This includes cache protocols, memory consistency models, atomic instructions and address translation including Translation Lookaside Buffer (TLB) support. These parts have already been implemented in the original VADL implementation [63] and will be added to OpenVADL.

Finally, the microarchitecture synthesis has to be extended to support all VADL logic elements such as reservation stations, reorder buffers, load/store queues, and fetch buffers. Further optimizations are necessary to generate competitive hardware.

8 CONCLUSION

This article presented VADL and its generators. The powerful language constructs in VADL allow concise and comprehensible specifications of the instruction set architecture, the microarchitecture and the application binary interface of processor architectures. Automatic generation of a toolchain including assemblers, compilers, linkers, functional and cycle accurate instruction set simulators and synthesizable hardware enables fast and efficient DSE of ASIPs. VADL has been successfully used to specify various common instruction set architectures like RISC-V, MIPS, Arm AArch32, Arm AArch64, Arm NEON, Texas Instruments TIC64x and to specify a large variation of scalar and out-of-order superscalar microarchitectures. The VADL research prototype is stable enough to employ the generated tools in the exploration of processor architectures. Additional information is available at the web page of [VADL](#).

ACKNOWLEDGMENTS

Part of this work was supported by a grant from Huawei. Hermann Schützenhöfer developed the first version of the simulator generator [111]. Alexander Graf developed the first version of the compiler generator [54]. Hristo Mihaylov developed the DTC simulator generator [90].

REFERENCES

- [1] 2002. IEEE Standard for Verilog Register Transfer Level Synthesis. *IEEE Std 1364.1-2002* (2002), 1–108. <https://doi.org/10.1109/IEEESTD.2002.94220>
- [2] 2004. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)* (2004), 1–118. <https://doi.org/10.1109/IEEESTD.2004.94802>
- [3] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), 1–1315. <https://doi.org/10.1109/IEEESTD.2018.8299595>
- [4] 2024. *Embench: A Modern Embedded Benchmark Suite*. <https://www.embench.org/> [Online; accessed 15-January-2024].
- [5] 2024. *Spike RISC-V ISA Simulator*. <https://github.com/riscv-software-src/riscv-isa-sim> [Online; accessed 15-January-2024].
- [6] 2025. *Codasip*. <https://codasip.com> [Online; accessed 7-February-2025].
- [7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [8] Gene Myron Amdahl, Gerrit Anne Blaauw, and Frederick Phillips Brooks. 1964. Architecture of the IBM System/360. *IBM Journal of Research and Development* 8, 2 (1964), 87–101. <https://doi.org/10.1147/rd.82.0087>
- [9] Hela Belhadj Amor, Carolyn Bernier, and Zdeněk Prikryl. 2022. A RISC-V ISA Extension for Ultra-Low Power IoT wireless Signal Processing. *IEEE Trans. Comput.* 71, 4 (2022), 766–778. <https://doi.org/10.1109/TC.2021.3063027>
- [10] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (April 1998), 17–20. <https://doi.org/10.1145/278283.278285>
- [11] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. <https://doi.org/10.1145/3290384>
- [12] Rafael Auler, Paulo Cesar Centoducatte, and Edson Borin. 2012. ACCGen: An Automatic ArchC Compiler Generator. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. 278–285. <https://doi.org/10.1109/SBAC-PAD.2012.33>
- [13] Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (2002), 59–67. <https://doi.org/10.1109/2.982917>
- [14] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. 2005. The ArchC architecture description language and tools. *International Journal of Parallel Programming* 33, 5 (2005), 453–484. <https://doi.org/10.1007/s10766-005-7301-0>
- [15] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzyniek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225. <https://doi.org/10.1145/2228360.2228584>
- [16] Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo, and Guido Araujo. 2004. Optimizations for compiled simulation using instruction type information. In *16th Symposium on Computer Architecture and High Performance Computing*. IEEE, 74–81. <https://doi.org/10.1109/SBAC-PAD.2004.28>
- [17] Robert Bedichek. 1990. Some Efficient Architecture Simulation Techniques, Winter 1990 USENIX Conference.
- [18] James R Bell. 1973. Threaded code. *Commun. ACM* 16, 6 (1973), 370–372. <https://doi.org/10.1145/362248.362270>
- [19] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. USENIX, 46. https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf
- [20] Lennart Beringer. 2022. *Functional Representations of SSA*. Springer International Publishing, Cham, 63–88. https://doi.org/10.1007/978-3-030-80515-9_6
- [21] Lorenzo Bettini. 2016. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition* (2nd ed.). Packt Publishing. <https://dl.acm.org/doi/10.5555/3074444>
- [22] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. <https://doi.org/10.1145/2024716.2024718>
- [23] Florian Brandner. 2009. *Compiler backend generation from structural processor models*. Ph. D. Dissertation. Technische Universität Wien. <https://repositum.tuwien.at/handle/20.500.12708/12553>
- [24] Florian Brandner, Dietmar Ebner, and Andreas Krall. 2007. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 13–22. <https://doi.org/10.1145/1289881.1289886>
- [25] Florian Brandner, Andreas Fellnhöfer, Andreas Krall, and David Riegler. 2009. Fast and accurate simulation using the LLVM compiler framework. In *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO*, Vol. 9. 1–6. https://www.complang.tuwien.ac.at/andi/papers/rapido_09.pdf
- [26] Florian Brandner, Nigel Horspool, and Andreas Krall. 2013. DSP instruction set simulation. *Handbook of Signal Processing Systems* (2013), 945–974. https://doi.org/10.1007/978-1-4614-6859-2_29
- [27] Florian Brandner, Viktor Pavlu, and Andreas Krall. 2012. Automatic generation of compiler backends. *Software Practice & Experience* 43, 2 (Feb. 2012), 207–240. <https://doi.org/10.1002/spe.2106>
- [28] Preston Briggs, Keith D Cooper, and L Taylor Simpson. 1997. Value Numbering. *Software: Practice and Experience* 27, 6 (1997), 701–724.
- [29] Harold W Cain, Kevin M Lepak, Brandon A Schwartz, and Mikko H Lipasti. 2002. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, Vol. 8. https://pharm.ece.wisc.edu/papers/caecw_2002_final.pdf

- [30] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation Via Coloring. *Computer Languages* 6, 1 (1981), 47–57. [https://doi.org/10.1016/0096-0551\(81\)90048-5](https://doi.org/10.1016/0096-0551(81)90048-5)
- [31] Bob Cmelik and David Keppel. 1994. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *SIGMETRICS Perform. Eval. Rev.* 22, 1 (May 1994), 128–137. <https://doi.org/10.1145/183019.183032>
- [32] John Cocke. 1970. Global Common Subexpression Elimination. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 20–24. <https://doi.org/10.1145/800028.808480>
- [33] John Cocke and Ken Kennedy. 1977. An Algorithm for Reduction of Operator Strength. *Commun. ACM* 20, 11 (Nov. 1977), 850–856. <https://doi.org/10.1145/359863.359888>
- [34] Keith D Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.
- [35] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. <https://doi.org/10.1145/115372.115320>
- [36] Robert BK Dewar. 1975. Indirect threaded code. *Commun. ACM* 18, 6 (1975), 330–331. <https://dl.acm.org/doi/pdf/10.1145/360825.360849>
- [37] Ralf Dreesen. 2011. *Generating Processors from Specifications of Instruction Sets*. Ph. D. Dissertation. University of Paderborn. <https://digital.ub.uni-paderborn.de/hsx/content/titleinfo/315766>
- [38] Ralf Dreesen. 2012. ViDL: A Versatile ISA Description Language. In *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*. 222–231. <https://doi.org/10.1109/ECBS.2012.49>
- [39] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.
- [40] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. [n. d.]. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages* (New York, NY, USA, 2013-10-28) (VML '13). Association for Computing Machinery, 1–10. <https://doi.org/10.1145/2542142.2542143>
- [41] M Anton Ertl. 2001. Threaded code variations and optimizations. In *EuroForth 2001 Conference Proceedings*. EuroForth, 49–55. <http://www.euroforth.org/ef01/ertl01.pdf>
- [42] M Anton Ertl and David Gregg. 2001. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings* 7. Springer, 403–413. https://link.springer.com/chapter/10.1007/3-540-44681-8_59
- [43] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 307–309. <https://doi.org/10.1145/1869542.1869625>
- [44] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. 2007. Ultra fast cycle-accurate compiled emulation of in-order pipelined architectures. *Journal of Systems Architecture* 53, 8 (2007), 501–510. <https://doi.org/10.1016/j.sysarc.2006.11.003>
- [45] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. 2006. Effective compiler generation by architecture description. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems* (Ottawa, Ontario, Canada) (LCTES '06). ACM, New York, NY, USA, 145–152. <https://doi.org/10.1145/1134650.1134671>
- [46] Andreas Fauth, Johan Van Praet, and Markus Freericks. 1995. Describing instruction set processors using nML. In *Proceedings the European Design and Test Conference. ED&TC 1995*. IEEE, 503–507. <https://doi.org/10.1109/EDTC.1995.470354>
- [47] Dirk Fischer, Jürgen Teich, Ralph Weper, Uwe Kastens, and Michael Thies. 2001. Design space characterization for architecture/compiler co-exploration. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Atlanta, Georgia, USA) (CASES '01). Association for Computing Machinery, New York, NY, USA, 108–115. <https://doi.org/10.1145/502217.502234>
- [48] Björn Franke. 2008. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*. ACM, 69–78. <https://doi.org/10.1145/1361096.1361109>
- [49] Christopher W. Fraser. 1991. A Retargetable Compiler for ANSI C. *SIGPLAN Notices* 26, 10 (Oct. 1991), 29–43. <https://doi.org/10.1145/122616.122621>
- [50] Norbert E. Fuchs. 1992. Specifications are (preferably) executable. *Software Engineering Journal* 7 (1992), 323–334. Issue 5. <https://doi.org/10.1049/sej.1992.0033>
- [51] Roberto Giorgi and Gianfranco Mariotti. 2019. WebRISC-V: a Web-Based Education-Oriented RISC-V Pipeline Simulation Environment. In *Proceedings of the Workshop on Computer Architecture Education*. ACM, 1–6. <https://doi.org/10.1145/3338698.3338894>
- [52] Shilpi Goel, Warren A Hunt, and Matt Kaufmann. 2017. *Engineering a formal, executable x86 ISA simulator for software verification*. Springer International Publishing, Cham, 173–209. https://doi.org/10.1007/978-3-319-48628-4_8
- [53] Ricardo E. Gonzalez. 2000. Xtensa: A Configurable and Extensible Processor. *IEEE Micro* 20, 2 (2000), 60–70. <https://doi.org/10.1109/40.848473>
- [54] Alexander Graf. 2021. *Compiler Backend Generation using the VADL Processor Description Language*. Master's thesis. Technische Universität Wien. <https://doi.org/10.34726/hss.2021.79221>
- [55] Peter Grun, Ashok Halambi, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alexandru Nicolau. 1998. *EXPRESSION: An ADL for system level design exploration*. Technical Report. University of California, Irvine. http://www.ics.uci.edu/~express/pubs/expr_tr.ps
- [56] Gary D Hachtel and Fabio Somenzi. 2005. *Logic synthesis and verification algorithms*. Springer Science & Business Media.
- [57] George Hadjiyiannis and Srinivas Devadas. 2003. Techniques for accurate performance evaluation in architecture exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11, 4 (2003), 601–615. <https://doi.org/10.1109/TVLSI.2003.812290>

- [58] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. 1997. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the 34th Annual Design Automation Conference (Anaheim, California, USA) (DAC '97)*. ACM, New York, NY, USA, 299–302. <https://doi.org/10.1145/266021.266108>
- [59] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. 2000. ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration. *Design Automation for Embedded Systems* 6 (2000), 39–69. <https://doi.org/10.1023/A:1008937425064>
- [60] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. 2008. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Design, Automation, and Test in Europe*. EDAA, 31–45. <https://dl.acm.org/doi/pdf/10.1145/307418.307549>
- [61] I.J. Hayes and C.B. Jones. 1989. Specifications are not (necessarily) executable. *Software Engineering Journal* 4 (1989), 330–339. Issue 6. <https://doi.org/10.1049/sej.1989.0045>
- [62] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier.
- [63] Simon Himmelbauer. 2024. *Atomic Instruction and Cache-Support for VADL*. Masters's Thesis. Technische Universität Wien. <https://doi.org/10.34726/hss.2024.113945>
- [64] Christoph Hochrainer and Andreas Krall. 2023. A Pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (Cascais, Portugal) (GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 29–41. <https://doi.org/10.1145/3624007.3624052>
- [65] Manuel Hohenauer and Rainer Leupers. 2009. *C Compilers for ASIPs*. Springer, New York, NY. <https://doi.org/10.1007/978-1-4419-1176-6>
- [66] Steve Holzner. 2004. *Eclipse*. O'Reilly Media, Inc.
- [67] Adam Husár, Miloslav Trmač, Jan Hranáč, Tomáš Hruška, and Karel Masařík. 2011. Automatic C Compiler Generation from Architecture Description Language ISAC. In *Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10) – Selected Papers (Open Access Series in Informatics (OASIs), Vol. 16)*, Ludek Matyska, Michal Kozubek, Tomas Vojnar, Pavel Zemcik, and David Antos (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 47–53. <https://doi.org/10.4230/OASIs.MEMICS.2010.47>
- [68] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. 2008. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the conference on Design, Automation and Test in Europe*. EDAA, 3–8. <https://doi.org/10.1145/1403375.1403380>
- [69] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, 209–216. <https://doi.org/10.1109/ICCAD.2017.8203780>
- [70] Ryan Kabrick. 2022. *Rapid Prototyping Framework for Hardware-Software Co-Design With Advanced Vector Architectures*. Master's thesis. University of Delaware. <https://udspace.udel.edu/handle/19716/31573>
- [71] Johannes Kappes, Robert Kunzelmann, Karsten Emrich, Conrad Foik, Daniel Müller-Gritschneider, and Wolfgang Ecker. 2023. Effective Processor Model Generation from Instruction Set Simulator to Hardware Design. In *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. 1–7. <https://doi.org/10.1109/NorCAS58970.2023.10305465>
- [72] David Keppel. 2009. How to Detect Self-Modifying Code During Instruction-Set Simulation. <http://www.xsim.com/papers/sim-smc-amasbt2009.pdf>
- [73] Paul Klint. 1981. Interpretation Techniques. *Software: Practice and Experience* 11, 9 (1981), 963–973. <https://doi.org/10.1002/spe.4380110908>
- [74] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy Code Motion. *SIGPLAN Not.* 27, 7 (July 1992), 224–234. <https://doi.org/10.1145/143103.143136>
- [75] Andreas Krall, Ivan Pryanishnikov, Ulrich Hirschnrott, and Christian Panis. 2004. xDSPcore: a compiler-based configurable digital signal processor. *IEEE Micro* 24, 4 (2004), 67–78. <https://doi.org/10.1109/MM.2004.40>
- [76] Daniel Kroening and Wolfgang J. Paul. 2001. Automated Pipeline Design. In *Proceedings of the 38th Annual Design Automation Conference (Las Vegas, Nevada, USA) (DAC '01)*. Association for Computing Machinery, New York, NY, USA, 810–815. <https://doi.org/10.1145/378239.379071>
- [77] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (Palo Alto, California) (CGO '04)*. IEEE Computer Society, USA, 75. <https://doi.org/10.1109/CGO.2004.1281665>
- [78] John Leidel, Ryan Kabrick, and David Donofrio. 2021. Toward an Automated Hardware Pipelining LLVM Pass Infrastructure. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 39–49. <https://doi.org/10.1109/LLVMHPC54804.2021.00010>
- [79] John D. Leidel, David Donofrio, and Frank Conlon. 2020. StoneCutter: a very high level instruction set design language. In *Proceedings of the 17th ACM International Conference on Computing Frontiers (Catania, Sicily, Italy) (CF '20)*. ACM, New York, NY, USA, 233–236. <https://doi.org/10.1145/3387902.3394029>
- [80] Allen Leung and Lal George. 1999. Static single assignment form for machine code. *ACM SIGPLAN Notices* 34, 5 (1999), 204–214. <https://doi.org/10.1145/301631.301667>
- [81] Rainer Leupers and Peter Marwedel. 1998. Retargetable Code Generation Based on Structural Processor Description. *Design Automation for Embedded Systems* 3, 1 (Jan. 1998), 75–108. <https://doi.org/10.1023/A:1008807631619>
- [82] Rainer Leupers and Peter Marwedel. 2001. *Retargetable Compiler Technology for Embedded Systems: Tools and Applications*. Springer Science+Business Media. <https://doi.org/10.1007/978-1-4757-6420-8>
- [83] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. 2015. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 256–267.

- <https://doi.org/10.1109/ISPASS.2015.7095811>
- [84] Guillem López-Paradis, Adrià Armejach, and Miquel Moretó. 2021. Gem5+ rtl: A framework to enable rtl models inside a full-system simulator. In *Proceedings of the 50th International Conference on Parallel Processing*. ACM, 1–11. <https://doi.org/10.1145/3472456.3472461>
 - [85] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020). <https://doi.org/10.48550/arXiv.2007.03152>
 - [86] Peter S Magnusson. 1997. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the 29th conference on Winter simulation*. 1093–1100. <https://dl.acm.org/doi/pdf/10.1145/268437.268745>
 - [87] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hällberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58. <https://doi.org/10.1109/2.982916>
 - [88] Peter Marwedel. 1984. The MIMOLA design system: Tools for the design of digital processors. In *21st Design Automation Conference Proceedings*. IEEE, 587–593. <https://doi.org/10.1109/DAC.1984.1585857>
 - [89] Kazutaka Matsuda and Meng Wang. 2013. FliPpr: A prettier invertible printing system. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings 22*. Springer, 101–120. https://doi.org/10.1007/978-3-642-37036-6_6
 - [90] Hristo Mihaylov. 2023. *Optimized processor simulation with VADL*. Master’s thesis. Technische Universität Wien. <https://doi.org/10.34726/hss.2023.102629>
 - [91] Christopher Mills, Stanley C Ahalt, and Jim Fowler. 1991. Compiled instruction set simulation. *Software: Practice and Experience* 21, 8 (1991), 877–889. <https://doi.org/10.1002/spe.4380210807>
 - [92] Prabhat Mishra and Nikil Dutt. 2008. *Processor Description Languages*. Elsevier. <https://doi.org/10.1016/B978-0-12-374287-2.X5001-0>
 - [93] Hanspeter Mössenböck. 2018. *The Compiler Generator Coco/R*. <https://ssw.jku.at/Research/Projects/Coco/>
 - [94] Steven Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan kaufmann.
 - [95] Michael Nestler. 2024. *Efficient parsing of OpenVADL*. Bachelor’s Thesis. Technische Universität Wien. <https://www.complang.tuwien.ac.at/vadl/papers/NestlerFinal.pdf>
 - [96] Julian Oppermann, Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Tammo Mürmann, Eyck Jentzsch, and Andreas Koch. 2024. Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (La Jolla, CA, USA) (ASPLOS ’24)*. ACM, New York, NY, USA, 591–606. <https://doi.org/10.1145/3620666.3651375>
 - [97] Preeti Ranjan Panda. 2001. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis (Montréal, P.Q., Canada) (ISSS ’01)*. Association for Computing Machinery, New York, NY, USA, 75–80. <https://doi.org/10.1145/500001.500018>
 - [98] Terence Parr and Kathleen Fisher. 2011. LL(*) the foundation of the ANTLR parser generator. *ACM SIGPLAN Notices* 46, 6 (2011), 425–436. <https://doi.org/10.1145/1993316.1993548>
 - [99] Terence J Parr and Russell W Quong. 1994. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *Compiler Construction: 5th International Conference, CC’94 Edinburgh, UK, April 7–9, 1994 Proceedings 5*. Springer, 263–277. https://doi.org/10.1007/3-540-57877-3_18
 - [100] David A. Patterson and John L. Hennessy. 2017. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
 - [101] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. 1999. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 933–938. <https://dl.acm.org/doi/pdf/10.1145/309847.310101>
 - [102] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. 2007. Compiler optimizations for processors with SIMD instructions. *Software: Practice and Experience* 37, 1 (2007), 93–113. <https://doi.org/10.1002/spe.751>
 - [103] Zdeněk Přikryl. 2014. Fast Simulation of Pipeline in ASIP Simulators. In *2014 15th International Microprocessor Test and Verification Workshop*. 10–15. <https://doi.org/10.1109/MTV.2014.18>
 - [104] Zdeněk Přikryl, Jakub Kroustek, Tomáš Hruška, Dušan Kolář, Karel Masařík, and Adam Husár. 2011. Design and Simulation of High Performance Parallel Architectures Using the ISAC Language. *GSTF Journal on Computing* 1, 2 (2011), 97–106. <http://dl6.globalsf.org/index.php/joc/article/download/894/2241>
 - [105] Zdeněk Přikryl, Karel Masařík, Tomáš Hruška, and Adam Husár. 2009. Fast cycle-accurate interpreted simulation. In *2009 10th International Workshop on Microprocessor Test and Verification*. IEEE, 9–14. <https://doi.org/10.1109/MTV.2009.11>
 - [106] Fabrice Rastello. 2016. *SSA-based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated.
 - [107] Tahiry Ratsimbahotra, Hugues Cassé, and Pascal Sainrat. 2009. A versatile generator of instruction set simulators and disassemblers. In *2009 International Symposium on Performance Evaluation of Computer & Telecommunication Systems*, Vol. 41. IEEE, 65–72. <https://ieeexplore.ieee.org/abstract/document/5224142>
 - [108] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. 2003. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *Proceedings of the 40th Annual Design Automation Conference*. ACM/IEEE, 758–763. <https://doi.org/10.1145/775832.776026>

- [109] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. 2002. Architecture implementation using the machine description language LISA. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15th International Conference on VLSI Design*. 239–244. <https://doi.org/10.1109/ASPDAC.2002.994928>
- [110] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 258–271. <https://doi.org/10.1145/3385412.3386024>
- [111] Hermann Schützenhöfer. 2020. *Cycle-Accurate simulator generator for the VADL processor description language*. Master’s thesis. Technische Universität Wien. <https://doi.org/10.34726/hss.2021.78460>
- [112] Tobias Schwarzingner. 2022. *Flexible generation of low-level developer tools with VADL*. Master’s thesis. Technische Universität Wien. <https://doi.org/10.34726/hss.2023.103246>
- [113] John Paul Shen and Mikko H Lipasti. 2013. *Modern processor design: fundamentals of superscalar processors*. Waveland Press.
- [114] Jeremy Singer. 2003. Static single information from a functional perspective. *Trends in Functional Programming* 4 (2003), 63–78.
- [115] Richard L. Sites. 1993. Alpha AXP Architecture. *Commun. ACM* 36, 2 (1993), 33–44. <https://doi.org/10.1145/151220.151226>
- [116] Richard Stallman et al. 2020. *Using the GNU Compiler Collection*. Gnu Press Boston.
- [117] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.
- [118] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33. <https://doi.org/10.1147/rd.111.0025>
- [119] Philipp Van Kempen, Mathis Salmen, Daniel Müller-Gritschneider, and Ulf Schlichtmann. 2024. Seal5: Semi-Automated LLVM Support for RISC-V ISA Extensions Including Autovectorization. In *2024 27th Euromicro Conference on Digital System Design (DSD)*. 335–342. <https://doi.org/10.1109/DSD64264.2024.00052>
- [120] Harry Wagstaff. 2015. *From High Level Architecture Descriptions to Fast Instruction Set Simulators*. Ph. D. Dissertation. The University of Edinburgh. <http://hdl.handle.net/1842/14162>
- [121] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. 2001. Hardware/software instruction set configurability for system-on-chip processors. In *Proceedings of the 38th Annual Design Automation Conference (Las Vegas, Nevada, USA) (DAC ’01)*. ACM, New York, NY, USA, 184–188. <https://doi.org/10.1145/378239.378460>
- [122] Peter Wegner. 1972. The Vienna Definition Language. *Comput. Surveys* 4, 1 (1972), 5–63. <https://doi.org/10.1145/356596.356598>
- [123] Clifford Wolf and Johann Glaser. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 6. <https://yosyshq.net/yosys/files/yosys-austrochip2013.pdf>
- [124] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. 2003. LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. In *Modular Programming Languages*, László Böszörményi and Peter Schojer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 192–201. <https://ssw.jku.at/Research/Papers/Woe03/WoeLoeMoe03.pdf>
- [125] Xin Xiao and Zhong Liu. 2023. ISADL: An Instruction Set Architecture Description Language for VLIW. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 92–99. <https://doi.org/10.1109/ICPADS60453.2023.00022>
- [126] Joshua J. Yi, Sreekumar V. Kodakara, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. 2005. Characterizing and comparing prevailing simulation techniques. In *11th International Symposium on High-Performance Computer Architecture*. 266–277. <https://doi.org/10.1109/HPCA.2005.8>
- [127] Joshua J. Yi and David J. Lilja. 2006. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *IEEE Trans. Comput.* 55, 3 (2006), 268–280. <https://doi.org/10.1109/TC.2006.44>
- [128] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. 1996. LISA – machine description language and generic machine model for HW/SW co-design. In *VLSI Signal Processing, IX*. IEEE, 127–136. <https://doi.org/10.1109/VLSISP.1996.558311>

APPENDIX A

RISC-V RV32I example

This appendix shows a complete formal specification of the RISC-V RV32I instruction set architecture. This specification contains all the necessary instructions and definitions to generate an ISS as it was used in the evaluation part of this article.

```

1  instruction set architecture RV32I = {
2
3      using Byte      = Bits < 8>           // 8 bit Byte
4      using Half      = Bits <16>          // 16 bit half word type
5      using Word      = Bits <32>          // 32 bit word type
6      using Index     = Bits < 5>           // 5 bit register index type for 32 registers

```

```

7  using SIntR      = SInt<32>           // register word signed type
8  using UIntR      = UInt<32>           // register word unsigned type
9  using UInt5      = UInt< 5>           // 5 bit unsigned shift ammount
10
11  [X(0) = 0]                           // register with index 0 always is 0
12  register file    X : Index -> Word   // integer register file with 32 registers of 32 bits
13  program counter PC : Word             // PC points to the start of the current instruction
14  memory           MEM : Word -> Byte   // byte addressed memory
15
16  format Rtype : Word =                 // Rtype register 3 operand instruction format
17  { funct7 : Bits<7>                   // [31..25] 7 bit function code
18    , rs2   : Index                     // [24..20] 2nd source register index / shamt
19    , rs1   : Index                     // [19..15] 1st source register index
20    , funct3 : Bits<3>                   // [14..12] 3 bit function code
21    , rd    : Index                     // [11..7] destination register index
22    , opcode : Bits<7>                   // [6..0] 7 bit operation code
23    , shamt  = rs2 as UInt               // 5 bit unsigned shift ammount
24  }
25  format Itype : Word =                 // Itype immediate instruction format
26  { imm     : Bits<12>                   // [31..20] 12 bit immediate value
27    , rs1   : Index                     // [19..15] source register index
28    , funct3 : Bits<3>                   // [14..12] 3 bit function code
29    , rd    : Index                     // [11..7] destination register index
30    , opcode : Bits<7>                   // [6..0] 7 bit operation code
31    , immS   = imm as SIntR             // sign extended immediate value
32  }
33  format Utype : Word =                 // Utype upper immediate instruction format
34  { imm     : Bits<20>                   // [31..12] 20 bit immediate value
35    , rd    : Index                     // [11..7] destination register index
36    , opcode : Bits<7>                   // [6..0] 7 bit operation code
37    , immU   = (imm as UIntR) << 12    // shifted unsigned immediate value
38  }
39  format Stype : Word =                 // Stype store instruction format
40  { imm     [31..25, 11..7]             // 12 bit immediate value
41    , rs2   [24..20]                   // 2nd source register index
42    , rs1   [19..15]                   // 1st source register index
43    , funct3 [14..12]                   // 3 bit function code
44    , opcode [6..0]                     // 7 bit operation code
45    , immS   = imm as SIntR             // sign extended immediate value
46  }
47  format Btype : Word =                 // Btype branch instruction format
48  { imm     [31, 7, 30..25, 11..8]      // 12 bit immediate value
49    , rs2   [24..20]                   // 2nd source register index
50    , rs1   [19..15]                   // 1st source register index
51    , funct3 [14..12]                   // 3 bit function code
52    , opcode [6..0]                     // 7 bit operation code
53    , immS   = (imm as SIntR) << 1     // sign extended and shifted immediate value immS
54  }
55  format Jtype : Word =                 // Jtype jump and link instruction format
56  { imm     [31, 19..12, 20, 30..21]    // 20 bit immediate value
57    , rd    [11..7]                     // destination register index
58    , opcode [6..0]                     // 7 bit operation code
59    , immS   = (imm as SIntR) << 1     // sign extended and shifted immediate value immS
60  }
61
62  model RtypeInstr (name: Id, op: BinOp, fu3: Bin, fu7: Bin, lhsTy: Id, rhsTy: Id) : IsaDefs = {

```

```

63   instruction $name : Rtype =                                     // 3 register operand instructions
64     X(rd) := ((X(rs1) as $lhsTy) $op (X(rs2) as $rhsTy)) as Bits
65   encoding $name = { opcode = 0b011'0011, funct3 = $fu3, funct7 = $fu7 }
66   assembly $name = (mnemonic, " ", register(rd), ",", register(rs1), ",", register(rs2))
67 }
68 model lShftInstr (name : Id, op : BinOp, funct3 : Bin, funct7 : Bin, lhsTy : Id) : IsaDefs = {
69   instruction $name : Rtype =                                     // shift immediate instructions
70     X(rd) := (X(rs1) as $lhsTy) $op shamt
71   encoding $name = { opcode = 0b001'0011, funct3 = $funct3, funct7 = $funct7 }
72   assembly $name = (mnemonic, " ", register(rd), ",", register(rs1), ",", decimal(rs2))
73 }
74 model lTypeInstr (name : Id, op : BinOp, funct3 : Bin, exTy : Id) : IsaDefs = {
75   instruction $name : ltype =                                     // immediate instructions
76     X(rd) := ((X(rs1) as $exTy) $op (immS as $exTy)) as Word
77   encoding $name = { opcode = 0b001'0011, funct3 = $funct3 }
78   assembly $name = (mnemonic, " ", register(rd), ",", register(rs1), ",", decimal(imm))
79 }
80 model UtypeInstr (name : Id, opcode : Bin, rhsEx : Ex) : IsaDefs = {
81   instruction $name : Utype =                                     // upper immediate instructions
82     X(rd) := $rhsEx
83   encoding $name = { opcode = $opcode }
84   assembly $name = (mnemonic, " ", register(rd), ",", hex(imm))
85 }
86 model lTypeInstr (name : Id, funct3 : Bin, memEx : CallEx, exTy : Id) : IsaDefs = {
87   instruction $name : ltype =                                     // load instructions
88     let addr = X(rs1) + immS in
89     X(rd) := $memEx as $exTy
90   encoding $name = { opcode = 0b000'0011, funct3 = $funct3 }
91   assembly $name = (mnemonic, " ", register(rd), ",", decimal(imm), "(", register(rs1), ")")
92 }
93 model StypeInstr (name : Id, funct3 : Bin, memEx : CallEx, exTy : Id) : IsaDefs = {
94   instruction $name : Stype =                                     // store instructions
95     let addr = X(rs1) + immS in
96     $memEx := X(rs2) as $exTy
97   encoding $name = { opcode = 0b010'0011, funct3 = $funct3 }
98   assembly $name = (mnemonic, " ", register(rs2), ",", decimal(imm), "(", register(rs1), ")")
99 }
100 model BtypeInstr (name : Id, relOp : BinOp, funct3 : Bin, lhsTy : Id) : IsaDefs = {
101   instruction $name : Btype =                                     // conditional branch instructions
102     if (X(rs1) as $lhsTy) $relOp X(rs2) then
103       PC := PC + immS
104   encoding $name = { opcode = 0b110'0011, funct3 = $funct3 }
105   assembly $name = (mnemonic, " ", register(rs1), ",", register(rs2), ",", decimal(imm))
106 }
107 model JLinkInstr (name : Id, iFormat : Id, reg : Ex, opcode : Encs, asm : Ex) : IsaDefs = {
108   instruction $name : $iFormat =                                   // jump and link (register)
109     let retaddr = PC.next in {
110       PC := (($reg + immS) as UInt) & 0xffff'fffe // $reg could be equal to X(rd)
111       X(rd) := retaddr // when rs1 is equal to rd
112     }
113   encoding $name = { $opcode }
114   assembly $name = (mnemonic, " ", register(rd), ",", decimal(imm), $asm)
115 }
116
117 $RtypeInstr (ADD ; + ; 0b000 ; 0b000'0000 ; Bits ; Bits ) // add
118 $RtypeInstr (SUB ; - ; 0b000 ; 0b010'0000 ; Bits ; Bits ) // subtract

```



```

119 $RtypeInstr (AND ; & ; 0b111 ; 0b000'0000 ; Bits ; Bits ) // and
120 $RtypeInstr (OR ; | ; 0b110 ; 0b000'0000 ; Bits ; Bits ) // or
121 $RtypeInstr (XOR ; ^ ; 0b100 ; 0b000'0000 ; Bits ; Bits ) // exclusive or
122 $RtypeInstr (SLT ; < ; 0b010 ; 0b000'0000 ; SInt ; SInt ) // set less than
123 $RtypeInstr (SLTU ; < ; 0b011 ; 0b000'0000 ; UInt ; UInt ) // set less than unsigned
124 $RtypeInstr (SLL ; << ; 0b001 ; 0b000'0000 ; UInt ; UInt5) // shift left logical
125 $RtypeInstr (SRL ; >> ; 0b101 ; 0b000'0000 ; UInt ; UInt5) // shift right logical
126 $RtypeInstr (SRA ; >> ; 0b101 ; 0b010'0000 ; SInt ; UInt5) // shift right arithmetic
127
128 $IShftInstr (SLLI ; << ; 0b001 ; 0b000'0000 ; UInt) // shift left logical immediate
129 $IShftInstr (SRLI ; >> ; 0b101 ; 0b000'0000 ; UInt) // shift right logical immediate
130 $IShftInstr (SRAI ; >> ; 0b101 ; 0b010'0000 ; SInt) // shift right arithmetic immediate
131
132 $ItypeInstr (ADDI ; + ; 0b000 ; SInt) // add immediate
133 $ItypeInstr (ANDI ; & ; 0b111 ; SInt) // and immediate
134 $ItypeInstr (ORI ; | ; 0b110 ; SInt) // or immediate
135 $ItypeInstr (XORI ; ^ ; 0b100 ; SInt) // exclusive or immediate
136 $ItypeInstr (SLTI ; < ; 0b010 ; SInt) // set less than immediate
137 $ItypeInstr (SLTIU ; < ; 0b011 ; UInt) // set less than immediate unsigned
138
139 $UtypeInstr (AUIPC ; 0b001'0111 ; PC + immU) // add upper immediate to PC
140 $UtypeInstr (LUI ; 0b011'0111 ; immU) // load upper immediate
141
142 $LtypeInstr (LB ; 0b000 ; MEM (addr) ; SIntR) // load byte signed
143 $LtypeInstr (LBU ; 0b100 ; MEM (addr) ; UIntR) // load byte unsigned
144 $LtypeInstr (LH ; 0b001 ; MEM<2>(addr) ; SIntR) // load half word signed
145 $LtypeInstr (LHU ; 0b101 ; MEM<2>(addr) ; UIntR) // load half word unsigned
146 $LtypeInstr (LW ; 0b010 ; MEM<4>(addr) ; SIntR) // load word
147 $StypeInstr (SB ; 0b000 ; MEM (addr) ; Byte ) // store byte
148 $StypeInstr (SH ; 0b001 ; MEM<2>(addr) ; Half ) // store half word
149 $StypeInstr (SW ; 0b010 ; MEM<4>(addr) ; Word ) // store word
150
151 $BtypeInstr (BEQ ; = ; 0b000 ; Bits) // branch equal
152 $BtypeInstr (BNE ; != ; 0b001 ; Bits) // branch not equal
153 $BtypeInstr (BGE ; >= ; 0b101 ; SInt) // branch greater or equal
154 $BtypeInstr (BGEU ; >= ; 0b111 ; UInt) // branch greater or equal unsigned
155 $BtypeInstr (BLT ; < ; 0b100 ; SInt) // branch less than
156 $BtypeInstr (BLTU ; < ; 0b110 ; UInt) // branch less than unsigned
157
158 $JLinkInstr (JAL ; Jtype ; PC ; opcode = 0b110'1111 ; "") // jump and link
159 $JLinkInstr (JALR ; Itype ; X(rs1) ; opcode = 0b110'0111, // jump and link register
160 func3 = 0b000 ; ("(", register(rs1), ")"))
161 }

```

Listing 34. Complete RISC-V RV32I ISA specification