TU WIEN Informatics

Generierung von Assemblern und Linkern in OpenVADL

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Benjamin Kasper
Matrikelnummer 12122530

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 27. September 2025

Benjamin Kasper	Andreas Krall

TU WIEN Informatics

Generation of Assemblers and Linkers in OpenVADL

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Benjamin Kasper
Registration Number 12122530

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, September 27, 2025

Benjamin Kasper	Andreas Krall

Erklärung zur Verfassung der Arbeit

Benjamin Kasper

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. September 2025

Benjamin Kasper

Kurzfassung

OpenVADL ist die Open-Source-Implementierung der Vienna Architecture Description Language. Dabei handelt es sich um eine Prozessorbeschreibungssprache, die eine schnelle Erkundung von Designvarianten in der Prozessorentwicklung ermöglichen soll, indem sie automatisch Artefakte wie Compiler, Simulatoren oder sogar Hardwarepläne erzeugt. Um Architekturen vollständig bewerten zu können, braucht man jedoch zusätzliche Werkzeuge wie Assembler und Linker. Zwar könnten diese auch manuell entwickelt werden, das würde jedoch der Anforderung schneller Iterationen nicht gerecht. In dieser Arbeit erweitern wir OpenVADL daher um die Möglichkeit, auch Assembler und Linker zu generieren. Dafür haben wir eine Assemblerspezifikation mit gut lesbarer Syntax entworfen und Assembler- und Linkergeneratoren auf Basis von LLVM implementiert. Die Ergebnisse unserer Auswertung zeigen, dass die Generierung der Werkzeuge leistungsfähig genug für Design-Space-Exploration ist und die erzeugten Assembler und Linker ähnlich performant wie ihre offiziellen Pendants im LLVM-Projekt sind.

Abstract

OpenVADL is the open-source implementation of the Vienna Architecture Description Language. The Vienna Architecture Description Language is a processor description language that aims to provide rapid design space exploration in processor development by generating artifacts such as compilers, simulators and even hardware schematics. To fully evaluate specified architectures, additional developer tools such as assemblers and linkers are needed. While manually crafting these tools is possible, it is not feasible when rapid iteration on designs is desired. In this work, we extend OpenVADL to support generation of assemblers and linkers. We designed an assembler specification grammar with an emphasis on syntax readability and implemented assembler and linker generators based on LLVM. Evaluation results show that the tool generation performance is suitable for design space exploration and that generated assemblers and linkers perform similarly to their LLVM upstream equivalents.

Contents

CHAPTER 1

Introduction

The Vienna Architecture Description Language (VADL) is a processor description language (PDL) aiming to enable fast design space exploration (DSE) in processor development [FHH+25b]. The core idea of VADL is to achieve this by automatically deriving the necessary tools from a concise specification. These include compilers, simulators, hardware, and also assemblers and linkers.

OpenVADL is an open-source[1] implementation of VADL improving on shortcomings of the original non-public implementation [FHH+25a]. Tobias Schwarzingers master's thesis 'Flexible generation of low-level developer tools with VADL' [Sch22] details the implementation of the assembler and linker generation in this original VADL implementation. Schwarzinger employs an LL(1) parsing algorithm in his assembler generator prototype. His evaluation shows that such an algorithm is in general sufficient to parse common assembly language constructs. While LL(1) algorithms are generally easier to implement than more involved parsing algorithms, this approach offloads complexity to the user specifying an assembly language and does therefore not capture VADLs spirit of concise specifications and fast DSE.

Therefore, in this work we aim to implement an improved assembler generator in the context of OpenVADL. By deploying a predicated-LL(1) algorithm the burden is moved from the user to the assembler generator. Additional improvements regarding specification readability should be made by improving the syntax of assembler specification constructs. Lastly, we plan to implement a linker generator in OpenVADL to enable the creation of executables for specified architectures.

In chapter 2 we give an overview of VADL, assemblers and linkers, recursive descent parsing and the LLVM project in the context of this work. Chapter 3 lists related work in the space of automatic assembler generation. In chapter 4 we present the

[1] OpenVADL is available at `https://github.com/OpenVADL/openvadl`

implementation details of automatic assembler and linker generation in OpenVADL. Chapter 5 discusses the performance of our implementation. Finally, chapter 6 lists possible future improvements and chapter 7 summarizes this work.

CHAPTER 2

Background

In this chapter we discuss the necessary preliminaries for this work. First, we introduce the relevant parts of VADL. Then we give an overview of assembling and linking. Lastly, we briefly explain the key technologies used to implement the assembler and linker generator in OpenVADL.

2.1 Vienna Architecture Description Language - VADL

A detailed introduction to VADL can be found in [FHH+25b]. For the purpose of assembler and linker generation we limit our overview of VADL to the instruction set architecture (ISA) and application binary interface (ABI) sections. Based on these two sections we will then expand on the assembly description (AD) section in chapter 4. The combination of these three sections defines the behavior of the generated assembler and linker.

The ISA section is the central part of a VADL specification. It defines behavior, encoding and different representations of instructions of an architecture. Listing 2.1 gives an example defining the `ADDI` instruction of the RISC-V architecture.

First, type aliases are defined to add semantic information to types (see lines 3 to 8). Next, a register file `X` is defined as a mapping from the `Index` type to the `Regs` type. As `Index` refers to the type `Bits<5>` and `Regs` to `Bits<32>`, this definition specifies that there are 32 registers of size 32 bits in the `X` register file (see line 11). Additionally, the annotation `zero : X(0)` declares the first register in the file as zero register (see line 10). A zero register always returns zero on read and ignores any writes to it.

Lines 13 to 20 define the `Itype` instruction format. A format defines the binary representation of an instruction. The `Itype` format can be used to define the so called ”immediate instructions” of RISC-V. These take 3 operands, namely a destination register (`rd`), a source register (`rs1`) and an immediate value (`imm`). The remaining fields

`funct3` and `opcode` differentiate the type of immediate instruction (e.g. ADDI, ANDI, ...). Additionally, the format contains the access function `immS` (line 19). It transforms the `imm` field to a 32 bit signed integer. As in this case for immediate instructions, fields often need to be transformed before their usage. Access functions provide a convenient use of transformed fields (as opposed to transforming the field at the site of usage).

```
1  instruction set architecture RV32I = {
2
3    constant Size = 32
4
5    using Byte    = Bits<  8 >
6    using Inst    = Bits< 32 >
7    using Regs    = Bits<Size>
8    using Index   = Bits<  5 >
9
10   [zero : X(0)]
11   register          X : Index -> Regs
12
13   format Itype   : Inst =
14     { imm        : Bits<12>
15     , rs1        : Index
16     , funct3     : Bits<3>
17     , rd         : Index
18     , opcode     : Bits<7>
19     , immS       = imm as SInt<Size>
20     }
21
22   instruction ADDI : Itype =
23     X(rd) := X(rs1) + immS
24   encoding ADDI = {opcode = 0b001'0011, funct3 = 0b000}
25   assembly ADDI = (mnemonic," ",register(rd),",",
26     register(rs1),",",sdec(imm))
27
28   pseudo instruction NOP =
29   {
30     ADDI{rd = 0 as Index, rs1 = 0 as Index, imm = 0 as Bits<12>}
31   }
32   assembly NOP = mnemonic
33 }
```

Listing 2.1: ISA specification of RISC-V ADDI and NOP instructions

Then the `ADDI` instruction is defined by three statements (see lines 22 to 26). The `instruction` keyword allows specifying the behavior of an instruction. Declaring the instruction to adhere to the `Itype` format allows defining the instructions behavior in terms of the format fields. Specifically, the `ADDI` instruction uses the access function

immS to add the value of field imm to the value of register file X at index rs1 and stores the result in X at index rd.

The encoding keyword allows setting fields of an instruction to constant values. In line 24 the opcode and funct3 fields are set to binary values 001 0011 and 000 respectively. As mentioned above, these values uniquely identify the ADDI instruction.

Finally, an assembly statement (see lines 25 and 26) is required to finish the instruction definition. It defines the textual representation of instructions in terms of its formats fields by a sequence of strings. Builtin functions help formatting the format fields as strings. The mnemonic function is replaced by the instructions name. register transforms the index into a register file to a string (eg. X(1) to "x1"). sdec transforms the immediate value imm into a signed decimal number representation.

VADL also allows the definition of pseudo instructions. Pseudo instructions are defined in terms of "normal" (or machine) instructions. As such the behavior and binary representation of pseudo instructions is implicitly defined by their machine instructions. Solely the string representation of pseudo instructions needs to be specified explicitly.

Lines 28 to 32 of Listing 2.1 show the definition of the "no operation" - NOP pseudo instruction. It is defined in terms of the ADDI machine instructions, where all parameters are 0 (= ADDI x0, x0, 0 in string representation). As NOP takes no operands, its string representation is just its name (see line 32).

```
1  instruction set architecture RV32I = { ... }
2
3  application binary interface ABI for RV32I = {
4    ...
5    alias register zero = X(0)
6    ...
7    alias register fp = X(8)
8    alias register s0 = X(8)
9    ...
10   alias register t6 = X(31)
11   ...
12 }
```

Listing 2.2: Register aliases in ABI specification of RISC-V

In the ABI of a VADL specification constructs needed for automatic compiler generation are defined. Among them are also alias register definitions. These introduce a name for an index into a register file. Listing 2.2 shows a few alias register definitions for the ISA of Listing 2.1. Lines 7 and 8 show that there can be multiple names for a register.

2.2 Assembler and Linker

2.2.1 Assembler

An assembler is a program that translates the textual representation of an architecture's instructions, its assembly language, to their binary representation - the machine code. To accomplish this, an assembler has two main tasks: parsing the assembly language and emitting the corresponding binary encoding. Statements in an assembly language can typically be categorized into the following constructs: labels, directives, machine instructions and pseudo instructions. A label marks a position in an assembly program. It is used to allow instructions to refer to segments of the assembly program.

Directives are commands that specify the behavior of an assembler. They either enable or disable options of the assembler or directly alter the produced binary stream. An example is the *.align <Byte>* directive which aligns the binary encoding of the instruction following the directive to an address that is a multiple of the specified byte number.

Machine instructions in an assembly language consist of a *mnemonic* followed by register- and / or immediate operands. Architectures define their registers and value ranges for immediate operands. After identifying an instruction by its mnemonic and operands, an assembler needs to verify that operands are valid. Register operands are expressions which need to match the registers defined by the architecture and immediate operands need to be within the valid value range. The final step is to emit the binary encoding corresponding to the validated instruction. Figure 2.1 shows the mapping of the RISC-V `ADDI` instructions operands to its binary format.

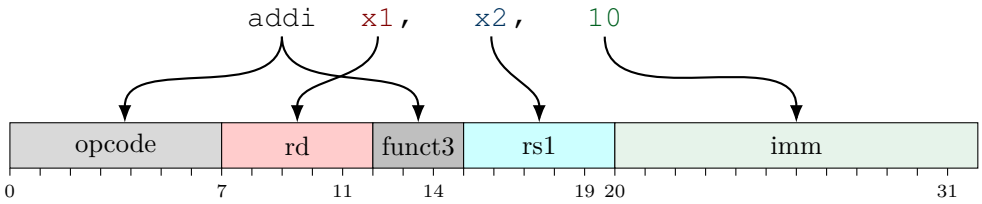

Figure 2.1: Mapping of RISC-V `ADDI` instruction to its binary format.

Pseudo instructions in an assembly program are indistinguishable from machine instructions. They also consist of a mnemonic followed by operands. The difference is that pseudo instructions do not have a binary encoding themselves, but rather are defined in terms of machine instructions (see definition in Listing 2.1). Therefore, an assembler needs to expand pseudo instructions to machine instructions and then emit their corresponding binary encoding. Figure 2.2 exemplifies this procedure with the RISC-V `NOP` pseudo instruction.

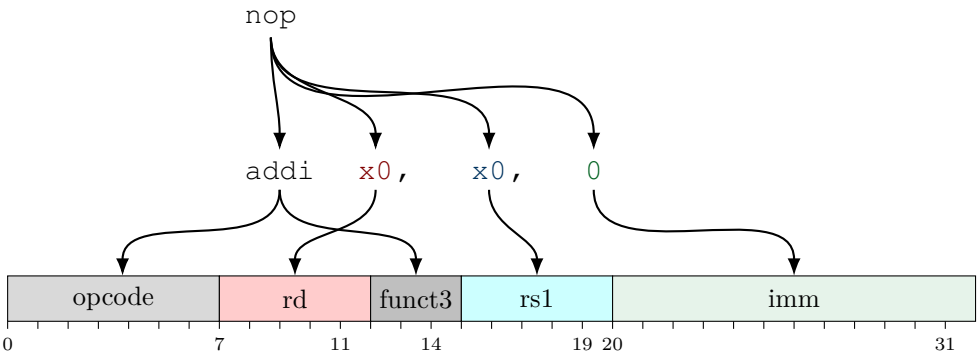

Figure 2.2: Expansion and mapping of RISC-V `NOP` pseudo instruction.

2.2.2 Linker

Source code of programs is usually split between multiple files to keep software maintainable. An assembler operates on a per file basis, so a process to combine all assembled sources of a program into an executable program is needed. A linker is a tool doing exactly this. It takes the binary representation of multiple source files and produces a single executable program.

An important concept for both assemblers and linkers are relocations. Relocations allow the splitting code of a single program into multiple files. Assemblers emit relocations to signal to the linker that an instruction refers to a label which address is unknown while assembling and needs to be determined in the linking process. This is the case whenever a code segment references a construct defined in another source file (e.g. calling a function defined in another file).

Once the linker combines all sources it needs to resolve all relocations. At this point addresses of labels can be determined and the instruction encodings referenced by the relocations get updated with the determined addresses. Some relocations require that addresses need to be further transformed before being written to the instruction encoding. In such cases the linker has to apply the transformation required by the relocation type.

Further, linkers also apply optimization techniques such as relaxation. Generally, the term relaxation describes a linker's ability to replace one instruction in the machine code with another instruction. The idea is to leverage architecture-specific information to improve either code size (e.g. choosing shorter instructions in a variable-length architecture) or runtime. A typical example for runtime improvement is branch relaxation. Take an architecture with two jump instructions as example. One is an absolute jump that allows jumps to any address, but takes two cycles to complete. The other is relative jump and only allows jumps to addresses within an 8 bit range relative to the current program counter, but takes just one cycle to complete. Now assume a program with a jump to an external symbol. As the assembler does not know whether the symbol's address is going to be within an 8 bit range, it emits the absolute jump instruction. In the linking

step, the linker resolves the symbols address and determines that the symbol is actually within an 8 bit range. Therefore, it can replace the absolute jump with the relative jump and reduce runtime by one cycle.

2.3 LLVM

The LLVM Compiler Infrastructure project was first released in 2004 as a tool to ease compiler creation [LA04]. Since then LLVM has developed into an umbrella project gathering a wide range of subprojects [llv25a]. Among them are the LLVM Machine Code Playground (LLVM-MC) and LLVM Linker (LLD) projects, which are the LLVMs assembler and linker respectively.

One of LLVMs core ideas is to provide as much functionality of a compiler toolchain as possible, while allowing comparatively easy specifications of new target architectures. For this reason OpenVADL utilizes LLVM in its compiler generator called LLVM Compiler Backend (LCB). As means of specification of new architectures serves LLVMs human readable file format TableGen [llv25b]. Therefore, one task of the LCB is to lower architectural information of a VADL specification to a TableGen specification.

In LLVM's spirit its assembler LLVM-MC also provides as much target independent functionality as possible. Recall from section 2.2.1 that an assembler has two main tasks: parsing an assembly language and producing its binary representation. For the former LLVM-MC provides utilities for parsing some constructs such as labels, directives and comments, while instructions are obviously architecture dependent and need to be implemented explicitly in the `MCAsmParser` component. For the latter LLVM-MC utililizes a TableGen specification and the instruction encoding information therein to create the `MCCodeEmitter` component.

As interface between these two components serves the internal data structure `MCInst`. The `MCAsmParser` creates one `MCInst` object per parsed instruction, containing the instruction type and the parsed operands. Operand values are wrapped in the `MCExpr` class, which provides utilities for dealing with the different types of operands. The `MCInst` objects are passed on to the `MCCodeEmitter` which calculates the binary encoding for an instruction and its operands according to the architectures TableGen specification.

LLVMs linker LLD handles the resolving of relocated symbols to addresses out-of-the-box. Any further architecture dependent transformations to resolved addresses and the routines to alter instruction encodings need to be implemented per target architecture.

2.4 LL(1) and predicated-LL(1) Parsing

As stated in chapter 1, the assembler generator in the original VADL implementation is limited to specifying assembly languages that can be parsed with an LL(1) parsing algorithm [Sch22]. LL parsers process their input from left to right while calculating

leftmost derivations. The number in parentheses refers to the number of tokens a parser considers when calculating derivations. So an LL(1) parser only ever considers one token at a time. This is sufficient for parsing common assembly languages, but can require complex specifications [Sch22].

The advantage of LL(1) parsers is that they are easy to implement, in particular as recursive descent parsers. An enhancement to LL(1) parsers are predicated-LL(1) parsers [PQD93]. Predicated refers to the addition of semantic predicates to a specification. This concept is used in popular parser generators such as ANTLR [PQ95] and Coco/R (called resolvers) [MLW18] [WLM03].

Semantic predicates are expressions evaluating to either true or false, inserted in a specification at decision points where multiple derivations are valid. An LL(1) algorithm would fail at this point, since it cannot determine which derivation to make. A predicated-LL(1) parser determines the derivation by evaluating the semantic predicate expressions in the order of their specification. The parser then simply chooses the first derivation where its semantic predicate evaluates to true.

Within semantic predicates, expressions evaluating more than the current token can be defined. This resolves the fundamental limitation of LL(1) parsers. In the application of parsing assembly languages this approach enables the parsing of abbreviated syntax variants. [Sch22] gives the example of the RISC-V `JALR` instruction. Listing 2.3 shows two forms of the same `JALR` instruction. The instruction in line 1 is a pseudo instruction where operand `rd` is always `x1` and `imm` is always 0. In line 2 all operands are specified explicitly.

```
1 jalr x2
2 jalr x1, x2, 0
```

Listing 2.3: Forms of the RISC-V `JALR` instruction

An assembler with the ability to parse both forms cannot be specified in the original VADL implementation. The reason for this is that the parser assigns parsed values to operands immediately. So at the point of parsing the first register operand, the LL(1) parser is not able to decide whether to assign the value to operand `rd` or operand `rs`. Whereas a predicated-LL(1) parser can check the next token in a semantic predicate. If there follows a ”,” after the register operand, the parser knows that `JALR` is in the form of line 2 and the register value needs to be assigned to `rd`. In the other case the parser knows that `rd` and `imm` are fixed and can assign the parsed register value to `rs`.

CHAPTER 3

Related Work

Automatic assembler and linker generation are most relevant in the context of architecture description language (ADL)s like VADL. [Sch22] describes the generation of assemblers and linkers in the original VADL implementation. Assembly syntax of instructions can be specified via a formal grammar within a VADL specification. For simple instruction syntaxes, grammar rules can also be inferred from instructions assembly printing definitions. From this formal grammar an assembler and linker based on LLVM are generated. A limitation of the generated assembler is its LL(1) parsing algorithm. One goal of this work is to eliminate this limitation in the new assembler generator of OpenVADL.

ArchC [RABA04] is an ADL that has been extended to support automatic generation of assemblers [BCR05] and linkers [CVDS06]. Both tools use information from an ArchC specification to generate target dependent libraries for the GNU Binutils software package. Compiling GNU Binutils with these libraries produces assemblers and linkers for architectures specified in ArchC.

[SWZR14] takes a similar approach utilizing GNU Binutils to automatically generate assemblers and disassemblers. It introduces GADL (GNU tool chain based ADL) as its specification language. As this project focuses solely on assembler and disassembler generation, its usage in DSE depends on other ADLs to generate simulators and further artifacts. This leads to redundant specifications in two ADLs or the need for a tool to generate one ADL from the other.

[BCR+08] generates assemblers, disassemblers and linkers also utilizing GNU Binutils. Specifications are given in an abstract model not tied to any ADL. Therefore, it faces the same issues regarding redundancies as [SWZR14]. To deal with this the authors give a formal definition of the abstract model in BNF to synthesize models from other ADLs.

LISA is an ADL aimed at modeling digital signal processors (DSP) [PHZM99]. It supports generation of artifacts like simulators, assemblers and linkers for such architectures [HNP+01]. Assembly syntax of instructions is specified via sequences of string constants

and operands in `SYNTAX` sections. Similarly, binary encoding is specified in `CODING` sections via binary constants and operands.

TIE (Tensilica Instruction Extension) is an ADL to configure extensions to the Xtensa processor [Gon00]. It is capable of automatically generating assemblers and linkers for new extensions to Xtensa. Like previously mentioned works it bases its tools on the GNU Binutils. Xtensa instructions follow one of six fixed formats. Therefore, binary encoding definitions for new instructions boil down to declaring the fixed opcode. Assembly syntax for new instructions is specified as sequence of operands.

UPFAST (University of Pittsburgh Flexible Architecture Simulation Tool) is a system focusing on generating simulators for microarchitectures [OG98]. Besides simulators UPFAST also generates assemblers and linkers. Because of the focus on microarchitecture, its ADL operates on a lower abstraction level. In the ADLs ISA section fields of instructions are specified with the `declare` keyword. Assembly syntax of instructions is specified by listing declared fields after the mnemonic within an `Instruction` block. Also within an `Instruction` block is the binary encoding specification with the `emit` keyword. There, a sequence of referenced fields and constants defines the instructions encoding.

The ADL NoGap (Novel Generator for ASIP) prioritizes generation of hardware descriptions, while also generating matching simulators and assemblers [KL09]. Its assembler generator is presented in [KLAL10]. The NoGap tool generates instruction definition files from a NoGap specification. The definition files are then loaded at runtime from the NoGap assembler to adjust it to specified architectures. Limitations of NoGap are that directives in assembly languages are not supported and that there is no automatic linker generation.

ISDL (Instruction Set Description Language) is an ADL aiming to aid DSE, particular in the space of VLIW architectures [HHD00]. To achieve this ISDL generates various tools such as simulators, compilers, assemblers and disassemblers. It is, however, not capable of generating linkers for specified architectures.

A recent ADL also focusing on VLIW architectures is ISADL [XL23]. Its main contribution is to automatically derive instruction encodings given a set of constraints regarding formats and number of instructions. Assemblers can be generated by specifying assembly syntax as regular expressions. To specify binary encoding a mapping from matched regular expressions to binary constants needs to be defined. Linker generation is not possible with ISADL.

CHAPTER 4

Implementation

4.1 Assembler Definition - Assembly Description in VADL

In this section we discuss the `assembly description` definition of a VADL specification. It specifies the behavior of the generated assembler. An `assembly description` depends on an ABI definition which in turn depends on an ISA definition (see lines 10 and 12 in Listing 4.1). This is because the assembler needs to be able to parse register aliases as defined in the ABI section (see Listing 2.2) and needs the instruction definitions of the ISA (see Listing 2.1) to emit correct binary encoding.

An `assembly description` consists of the mandatory subsection `grammar` as well as the optional subsections `modifiers` and `directives`. Additionally, the `assembly description` can be annotated to adjust the generated assemblers behavior in more detail and allows definition of common definitions to be used within the `grammar` subsection. The following subsections explain all of these components in detail.

4.1.1 Modifiers

Modifiers are transformation functions in an assembly language. When applied to constant operands the transformation can be done by the assembler. If such a transformation is applied to a symbol operand, the assembler emits a relocation to signal to the linker that it needs to apply the transformation after determining the symbols address. As the assembler needs to know which relocation to emit when encountering a certain modifier, this information needs to be specified in the `modifiers` subsection. It allows defining a mapping between strings to be used in the assembly language and relocations defined in the ISA section.

The example in Listing 4.1 shows an ISA definition and an assembly description refering to this ISA (via the ABI section). The relocations in the ISA define the transformations functions to be applied by the assembler or linker. The `hi` and `lo` relocations both take

```
instruction set architecture RV32I = {
  ...
  relocation hi( symbol : Bits<32> ) -> UInt<20>
    = ( ( symbol + 0x800 as Bits<32> ) >> 12 ) as UInt<20>

  relocation lo( symbol : Bits<32> ) -> SInt<12>
    = symbol as SInt<12>
}

application binary interface ABI for RV32I = { ... }

assembly description ASM for ABI = {
  ...
  modifiers = {
    "hi" -> ISA::hi,
    "lo" -> ISA::lo
  }
  ...
}
```

Listing 4.1: Example of modifier definition

a `Bits<32>` value as input and transform them to an `UInt<20>` and `SInt<12>` value respectively. [1]

The `modifiers` subsection then defines the mapping from strings in the assembly language to user defined relocations. So should the assembler encounter the modifier ”hi” applied to a constant operand, it applies the transformation function `hi` before emitting the binary encoding for the operand. To further illustrate take the two instructions in Listing 4.2 as example. As `%hi(0x1234)` is equal to `0x1` the assembler produces the same binary encoding for both instructions.

```
ADDI x1, x2, %hi(0x1234)
ADDI x1, x2, 0x1
```

Listing 4.2: Modifier in assembly language

In case a modifier is applied to a symbol operand, the assembler emits a relocation. This signals to the linker that the transformation needs to be applied after resolving the symbols address.

[1] More precisely: Applying `hi` returns the ”upper” 20 bit of the 32 bit input value and applying `lo` returns the lower 12 bit of the 32 bit input value.

4.1.2 Directives

Directives are commands within assembly programs that specify an assemblers behavior. OpenVADL offers a set of about 170 pre-defined directives available in common assemblers. [2] The `directives` subsection allows the definition of custom directive names by defining a mapping from a new name to an OpenVADL directive.

In Listing 4.3 three directives are renamed. First, `.align` is mapped to `ALIGN_POW2`. This defines the `.align` directive to align instructions at addresses that are multiples of two to the power of the parameter of the directive (e.g. `.align 3` aligns instructions at addresses which are multiple of 8). Further, the two directives `.quad` and `.4byte` are renamed to `.dword` and `.word` respectively.

```
1 assembly description ASM for ABI = {
2   ...
3   directives = {
4     ".align" -> ALIGN_POW2,
5     ".dword" -> QUAD,
6     ".word" -> BYTE4
7   }
8   ...
9 }
```

Listing 4.3: Example of directive renaming

4.1.3 Grammar

The `grammar` subsection is the core of the assembler specification. It defines the assembly language for the specified architecture in terms of a formal language. More precisely, a `grammar` section consists of a set of grammar rules which define the syntax of the textual representation of instructions. An observant reader may notice that this leads to a certain degree of redundancy with the `assembly` definitions of instructions in the ISA section (see 2.1). This slightly violates the VADL principle of concise specifications. Nevertheless, this compromise was made to allow greater flexibility when defining the assembly language. An example where the advantage of flexibility applies are architectures like RISC-V where instructions can have multiple valid textual representations. In this case the `assembly` definition of the ISA is limited to defining one syntax, while the `grammar` section allows defining constructs such that the assembler can parse all valid textual representations.

As already mentioned the `grammar` is made up of grammar rules in an EBNF like syntax. A rule has a name, an optional `AsmType` and a rule body, made up of grammar elements.

[2] A list of all supported directives can be found in the OpenVADL reference manual at `https://openvadl.github.io/openvadl/refmanual.html`

Listing 4.4 shows rules with basic grammar elements in their bodies. The rules in the example expect the following input:

- `LiteralA`: Expects ”a”.
- `Sequence`: Expects ”a” followed by ”b”.
- `Alternative`: Expects one of ”a”, ”b” or ”c”.
- `RuleReference`: Expects ”a” (as this is the body of `LiteralA`).
- `Optional`: Expects ”a” followed by zero or one ”b”.
- `Repetition`: Expects ”a” followed by zero or more ”b”.

```
1 grammar = {
2   LiteralA: "a" ;
3   Sequence: ("a" "b") ; // parentheses are optional
4   Alternative: "a" | "b" | "c" ;
5   RuleReference: LiteralA<> ; // angle brackets are optional
6   Optional: "a" [ "b" ] ;
7   Repetition: "a" { "b" } ;
8 }
```

Listing 4.4: Basic grammar elements

Out of these grammar elements *Alternatives*, *Optionals* and *Repetitions* can lead to LL(1) conflicts as shown in Listing 4.5. Line 2 shows a rule body where both alternatives expect the token ”a” as first element. This constitutes an LL(1) conflict because upon reading input ”a” an LL(1) parser cannot decide which of the two is the desired alternative to derive. Often such a conflict can be solved by rewriting the grammar rule. In the example the ”a” could be extracted to be outside of the alternatives to solve the conflict. But in many cases rewriting grammar rules to be LL(1) conform decreases the overall readability of a specification. Therefore, OpenVADL introduces semantic predicates to deal with LL(1) conflicts. Semantic predicates are expressions of the OpenVADL type system specified exactly at the first location of the conflict.

The grammar rule definition in Line 4 to 8 of Listing 4.5 shows the application of a semantic predicate to solve the conflict between alternatives starting with ”a”. The `?(...)` syntax specifies a semantic predicate at the beginning of the first alternative. Within the semantic predicate the special builtin function `LaIdEq` is called. `LaIdEq` is short for *Lookahead Identifier Equals* and allows to consider input tokens ahead of the current input token. It takes two arguments, where the first is a natural number `n` specifying the token to consider is `n` tokens ahead of the current token. `LaIdEq` returns

```
1  InvalidAlternative :
2    "a" "b" | "a" "c" ; // LL(1) conflict
3
4  ValidAlternative :
5    ?( LaIdEq(1, "b") ) // check for "b" in semantic predicate
6    "a" "b"
7    | "a" "c"
8  ;
9
10 ValidOptional :
11   [?( LaIdEq(1, "b") ) "a" "b"] "a" "c" ;
12
13 ValidRepetition :
14   {?( LaIdIn(1, "b", "d") ) "a" "b"} "a" "c";
```

Listing 4.5: LL(1) conflicts and semantic predicates

true if its second argument is equal to this specified token and false otherwise. So the semantic predicate in Line 5 checks whether the next token is equal to ”b”. If this is the case the expression evaluates to true and the first alternative is derived. If the next token is not equal to ”b” then the second alternative is derived.

Listing 4.5 also shows the cases where LL(1) conflicts arise with *Optional* and *Repetition* grammar elements. Similarly to the case with alternatives the conflicts are resolved with semantic predicates at the location of the first conflicting construct. In line 14 the second special builtin function `LaIdIn` is used. Like `LaIdEq` it considers the lookahead token at location of its first argument. But unlike `LaIdEq`, `LaIdIn` takes an arbitrary amount of string arguments after the first argument. If any of these strings are equal to the considered token, `LaIdIn` returns true.

Lastly, the burden of identifying LL(1) conflicts is not offloaded to the specifier. OpenVADL checks each grammar rule for LL(1) conflicts using First- and Follow-Sets. In case there are any conflicts, a detailed error message with exact error locations in the specifications is printed. The specifier can then decide to either rewrite the conflicting grammar rules or apply semantic predicates at the reported locations.

Before discussing the remaining grammar elements, it is necessary to consider AsmTypes, the special type system for grammar elements. AsmTypes are introduced as a means to give semantic meaning to parsed input. For example the assembler needs to differentiate whether a parsed instruction operand is supposed to be a register or a symbol operand.

Listing 4.6 shows a simple example of using AsmTypes in grammar rules. In the rule `RegisterStrings` the usage of `@string` explicitly marks the alternatives element and the rule itself to be of AsmType `@string`. As declaring this each and every time would be quite tedious, OpenVADL infers AsmTypes where possible. In the example

```
1 grammar = {
2   RegisterStrings @string : "x1" @string | "x2" @string ;
3
4   RegisterRule : RegisterStrings @register ;
5 }
```

Listing 4.6: Basic AsmType usage

of rule `RegisterStrings`, actually none of the `@string` declarations are needed as they can be inferred automatically. String literals like ”x1” and ”x2” are always of AsmType `@string`, therefore the whole alternatives grammar element also is of AsmType `@string`. As then the rule body is of AsmType `@string` the rule itself is inferred to be of AsmType `@string`.

AsmType	Description
`@constant`	Represents an integer value.
`@expression`	Represents a complex expression for an immediate operand. The `Expression` default rule is of this type.
`@instruction`	Represents an entire machine or pseudo instruction. It needs to consist of at least the mnemonic operand.
`@modifier`	Represents a modifier defined in the `modifiers` mappings of the assembly description.
`@operand`	Represents an instruction operand, which is used to build an instruction in the parser.
`@operands`	Represents a sequence of `@operand`.
`@register`	Represents a register of the ISA or register alias of the ABI.
`@statements`	Represents a sequence of `@instruction`, where each instruction is followed by an EOL.
`@string`	Represents a sequence of characters. Most terminal rules are of this type.
`@symbol`	Represents a reference to a symbol, such as an assembly label.
`@void`	Represents the empty type. For rules like EOL.

Table 4.1: Overview of AsmTypes

Table 4.1 lists an overview of all AsmTypes and their semantics. In order to create grammar elements of AsmTypes like `@register` or `@instruction` AsmType casting needs to be applied. Line 4 in Listing 4.1 shows an example of such an operation. In the body of `RegisterRule` the rule `RegisterStrings` is invoked. As previously described, `RegisterStrings` is of AsmType `@string`. By adding `@register` after the rule invocation it is cast to this AsmType. Not all AsmTypes can be arbitrarily cast from one to another, a list of valid casts is given in Table 4.2. Note that valid in

this context refers to assembler build time. For example it is generally possible to cast `@string` to `@register`. But this cast will lead to an assembler runtime error, should the parsed string not be an actual register specified in the ISA or an register alias of the ABI.

Source AsmType	Target AsmType	Semantics
`@instruction`	`@statements`	Create a sequence of statements containing a single instruction.
`@operands`	`@instruction`	Create an instruction from a sequence of operands.
`@operand`	`@instruction`	Create an instruction from a single operand.
`@register`	`@operand`	Wraps register in an operand.
`@constant`	`@operand`	Wraps constant in an operand.
`@string`	`@operand`	Wraps string in an operand.
`@expression`	`@operand`	Wraps expression in an operand.
`@symbol`	`@operand`	Wraps symbol in an operand.
`@modifier` with `@expression`	`@operand`	Create a new expression from a modified expression and wrap in an operand.
`@constant`	`@register`	Interpret integer as register index.
`@string`	`@register`	Interpret string as register name.
`@string`	`@modifier`	Create modifier from a string. The modifier must be defined in the `modifiers` section.
`@string`	`@symbol`	Interpret string as symbol name.
any	`@void`	Drops all data.

Table 4.2: Valid AsmType casts

So far, examples only contained fixed string literals (e.g. ”a”) as basic building blocks of grammar rules. Since an assembler certainly needs to parse arbitrary immediate values (which include both symbols and numeric values), further primitives that allow specifying such values in grammar rules are needed. To solve this issue, OpenVADL contains a set of terminal default grammar rules. [3] Most notable of these builtin rules are `INTEGER` and `IDENTIFIER`. The `INTEGER` terminal rule is of AsmType `@constant` and allows parsing of numbers in decimal, binary (prefixed with 0b) and hexadecimal format (prefixed with 0x). The `IDENTIFIER` terminal rule is of AsmType `@string` and allows parsing of strings starting with a letter followed by alphanumeric characters.

In addition to terminal default grammar rules, OpenVADL also defines a set of non-terminal default grammar rules. An overview of these is given in table 4.3. Of the non-terminal default rules *Instruction* and *Statement* have a special status. After parsing all grammar rules defined in the `grammar` subsection OpenVADL defines the *Instruction* rule as alternative over all defined rules with the AsmType `@instruction`. The

[3] The full set of terminal default grammar rules is specified in the OpenVADL reference manual at https://openvadl.github.io/openvadl/refmanual.html

Statement rule is then defined as the *Instruction* rule followed by an End-of-Line token. As such *Statement* serves as the entry point for the assemblers parser. All other non-terminal default rules in 4.3 expand on the terminal default rules to capture common assembly language constructs. They have no special implementation and could easily be explicitly specified in the `grammar` section, but are pre-defined for convenience. Lastly, non-terminal default grammar rules can be overwritten by specifying a rule with the exact same name in the `grammar` section.

Rule	AsmType	Description
Expression	`@expression`	An expression can be a signed integer, a complex expression (e.g., 2 + 3) or a symbol reference (e.g., .foo).
Identifier	`@string`	Identifier allows parsing of any string.
Immediate-Operand	`@operand`	ImmediateOperand is a convenience rule consisting of an expression with a cast to `@operand`.
Instruction	`@instruction`	The instruction default rule is an alternative over all grammar rules with the type @instruction.
Label	`@symbol`	Label is a symbol reference (e.g., .foo).
Natural	`@constant`	Natural is an unsigned integer number.
Integer	`@constant`	Integer is a signed integer number.
Register	`@register`	Register is any of the registers defined in the ISA or an register alias of the ABI.
Statement	`@instruction`	Statement is the Instruction default rule followed by an End-Of-Line token.

Table 4.3: Non-terminal default grammar rules

The final grammar elements to be discussed are *Attributes* and *Local Variables*. Attributes constitute the connection between the assembly grammar and instructions of the ISA section. To specify an instruction in the grammar, grammar elements of type `@operand` are assigned to attributes which correspond to the instructions format fields or field access functions. Whether to use a format field or its field access function as attribute depends on the ”view” in which a value is expected in the assembly language. As explained in section 2.1 a field access function transforms the fields value. We call the original value ”field view” and the transformed value its ”field access view”. If the field view is expected in the assembly language, a parsed operand can be emitted as binary without further adjustment. But in the case an instruction in the assembly language expects the field access view as operand, the value needs to be adjusted to fit the field view before emitting its binary encoding. For this the inverse field access function has to be applied. For simple field access functions this inversion is automatically derived by OpenVADL. For complex cases it needs to be explicitly specified as *encoding* function in the format definition.

Listing 4.7 exemplifies how attributes and local variables are used to define a grammar

```
1  grammar = {
2    AddiInstruction @instruction:
3      var tmp = null @operand
4      mnemonic = "ADDI" @operand
5      tmp = Register @operand ","
6      rs1 = Register @operand ","
7      immS = ImmediateOperand
8      rd = tmp
9    ;
10 }
```

Listing 4.7: Attributes and local variables

rule for the `ADDI` instruction defined in Listing 2.1. In line 3 the local variable `tmp` of type `@operand` is declared. Local variables are used to hold results of grammar elements which should be parsed at this point, but assigned to attributes at a later point in the grammar rule. Like in line 4, any instruction definition must first assign the special `mnemonic` attribute. This is a limitation given by the instruction matching implementation, which is responsible for determining the instruction definition of the ISA section corresponding to the mnemonic. After the `mnemonic` attribute the definitions for the instructions operands follow. In line 5 the result of parsing the first register operand is stored in local variable `tmp`. By adding "," at the end of line 5 the expectation of a comma character after the first register operand is defined. Similarly, line 6 defines the second operand to be a register operand followed by a comma character. The operand result is directly stored to the `rs1` format field attribute. For the third operand the non-terminal default grammar rule `ImmediateOperand` is used. Differently from the first two register operands its result is not assigned to a format field attribute, but to the field access function attribute `immS`. Finally, the result for the first operand, held in the `tmp` local variable, is assigned to the format field attribute `rd`.

For the repetition grammar element a special syntax allows cumulative assignments to attributes. As elements within a repetition block can be parsed multiple times all of the results can be collected in lists of parsed values. This mechanism is limited to the attributes for sequence AsmTypes, namely `@operands` and `@statements`. Listing 4.8 shows an example with the `@statements` AsmType.

4.1.4 Annotations and Common Definitions

The assembly description definition allows two annotations to configure the generated assembler. Listing 4.9 shows them in lines 1 and 2. First, the `comment string` annotation allows defining a custom string to be used as token to signal comments in an assembly program. When encountering this token the assembler ignores the rest of the current program line and continues with the statement in the next line. Second, the

```
grammar = {
  Statements:
    stmts = Statement @statements
    {
      stmts += Statement
    }
  ;
}
```

Listing 4.8: Cumulative assignment in repetition

`case sensitive` annotation configures whether the assembler should consider the case of letters when comparing strings. Listing 4.9 show the default values "#" and `false`, which are applied if an annotation is not explicitly specified.

```
[ comment string: "#" ]
[ case sensitive: false ]
assembly description ASM for ABI = {

  function x1_string -> String = "x1"

  using AsmInt = SInt<64>
  constant one = 1
  function plus_one(x: AsmInt) -> AsmInt = x + one

  grammar = {
    SomeInstruction @instruction:
      rd = (x1_string @register) @operand
      rs = Register @operand ","
      imm = plus_one<Integer> @operand
    ;
  }
}
```

Listing 4.9: Annotations and common definitions

To allow arbitrary transformations of parsed values in the assembler the assembly description permits the definition of functions, constants and using statements. While constants and using statements can only be used to define other common definitions, functions can be used in the body of grammar rules. The syntax for function calls within grammar rules is the functions name followed by its parameters separated by commas within angle brackets. In case a function takes no parameters, the angle brackets are optional and can be omitted for brevity.

Listing 4.9 shows such common definitions within an assembly description. Line 5 defines a parameterless function always returning the string ”x1”. Line 7 introduces the new name `AsmInt` for the type `SInt<64>`. Line 9 declares the function `plus_one` which takes one parameter and adds the constant defined in Line 8 to it.

The defined functions are used to specify grammar rule `SomeInstruction` (see Line 13 to 17). In Line 14 the call to function `x1_string` and subsequent casts to `@register` and `@operand` set the operand `rd` to register x1. Line 16 shows the passing of a parameter to a function using the angle brackets syntax. The builtin grammar rule `Integer` is called as argument for the function `plus_one`. This leads to the assembler expecting to parse an integer value and then calling the function with the parsed value as argument.

To use functions in grammar rules the VADL type of functions and AsmTypes at the site function usage need to match. This applies when passing parsing results as function arguments and when using function return values in grammar rules. To check the matching, supported AsmTypes are converted to their VADL type equivalent. A mapping of supported AsmTypes to their equivalent VADL types is given in Table 4.4.

Take function `plus_one` in Listing 4.9 as example. Its parameter and return type is `AsmInt`, which refers to the `SInt<64>` type. When called in Line 15, its argument is the `Integer` builtin grammar rule. This rule is of AsmType `@constant` and its equivalent VADL type `SInt<64>`. Therefore, the grammar rule can be used as argument for the function. The same applies for the return type. `plus_one` returns an `SInt<64>` which equals the `@constant` AsmType and can therefore be casted to an `@operand` AsmType.

However, an exact type match like in the example is not necessary. For function parameters it is sufficient if the VADL type of the passed argument can be implicitly cast to the VADL type of the parameter. [4]

AsmType	VADL type
@constant	SInt<64>
@string	String
@void	Void

Table 4.4: Mapping of AsmType to matching VADL type

4.2 Assembler and Linker Generation

The previous section detailed how assemblers can be specified in OpenVADL. In this section we discuss the components in OpenVADLs architecture responsible for generating assemblers and linkers from such specifications.

[4] For details on VADLs type system and implicit casting refer to the OpenVADL reference manual at `https://openvadl.github.io/openvadl/refmanual.html`

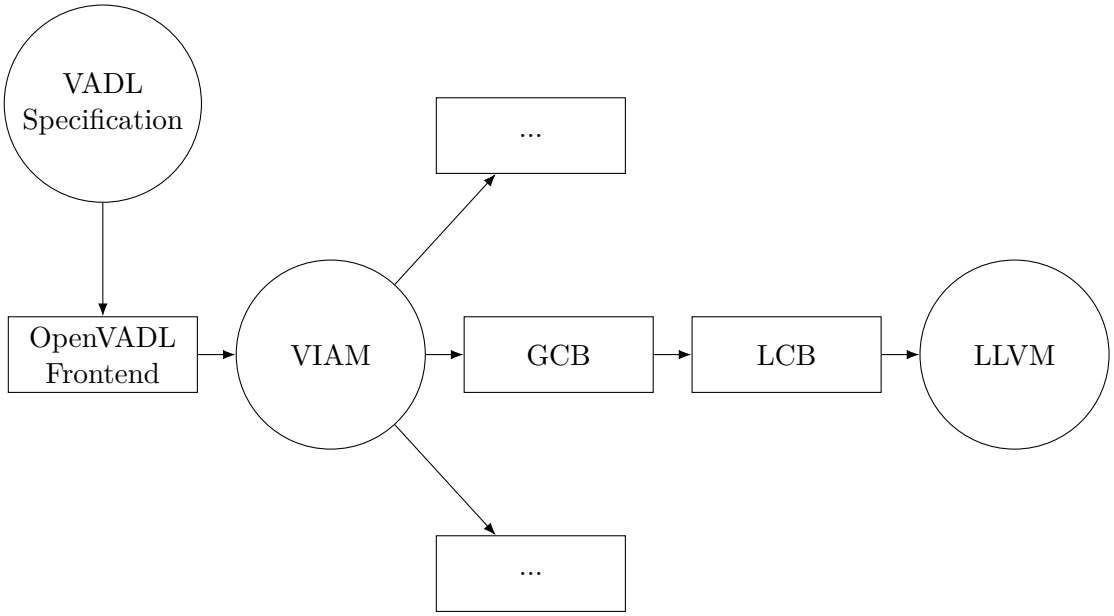

Figure 4.1: Coarse-grained overview of OpenVADLs architecture

Figure 4.1 shows a coarse-grained overview of OpenVADLs architecture. A specification first passes through the *Frontend* component. The Frontend performs lexical and syntactic analysis with its parser based on the parser generator Coco/R [MLW18]. Semantic analysis is implemented in the *Typechecker* module. As part of semantic analysis the *AsmLL1Checker* checks for LL(1) conflicts within the grammar subsection of an assembly description. Once the validity of a specification is verified by the Frontend, it is lowered to OpenVADLs intermediate representation - the VADL Intermediate Architecture Model (VIAM). Based on the VIAM, various generator components create artifacts like compilers and simulators. Two of these components are the Generic Compiler Backend (GCB) and the LCB, which was already mentioned in section 2.3. Together they constitute the OpenVADL compiler generator. At the time of writing both are being actively developed as part of a masters thesis and first results have been published [FHH+25a].

The GCB operates on the VIAM and contains analysis passes that create data structures in preparation for the subsequently following LCB. Based on the results of the GCB the LCB creates C++ source code files containing callbacks with target architecture dependent implementations to be compiled with LLVM in order to create an compiler for the specified architecture. Since LLVM also includes projects for assemblers (LLVM-MC) and linkers (LLD) following the same modular approach, we decided to implement OpenVADLs assembler and linker generator in the context of its LCB component. This takes advantage of the close relationship between compilers and assemblers in LLVM, as certain target dependent artifacts can be used by both developer tools.

4.2.1 Assembler

LLVM-MCs assemblers architecture reflects the two main tasks of assemblers, parsing instructions and emitting machine code. Figure 4.2 shows the architecture in the context of OpenVADL. Components in blue contain target architecture specific implementations. Arrows represent the calling structure between components when issuing an assembly command to the generated assembler. In the following paragraphs we will discuss how OpenVADL generates the depicted components.

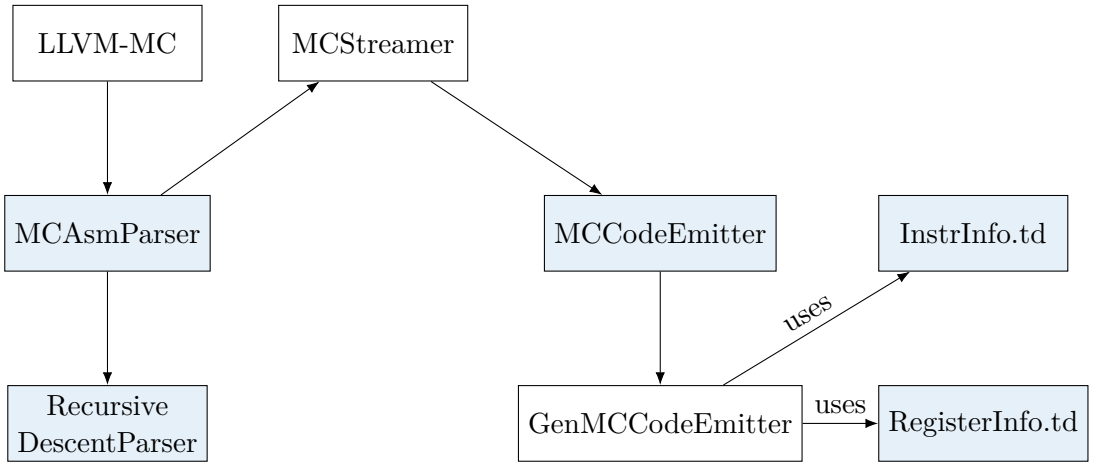

Figure 4.2: OpenVADL LLVM-MC assembler architecture

MCAsmParser

Parsing of common assembly language constructs like labels and directives is handled by target independent implementations given by the LLVM-MC framework. The parsing of instructions is split into two methods - `ParseInstruction` and `MatchAndEmitInstruction` - that need to be overwritten with target dependent implementations. Upon encountering an instruction in the assembly language input, LLVM-MC calls `ParseInstruction`, expects it to parse a single instruction statement and return a list of parsed instruction operands as result. OpenVADL redirects this call to the generated RecursiveDescentParser, which is described below. Once the operands are obtained, they are passed on to the `MatchAndEmitInstruction` method. Based on the first operand, the instructions mnemonic, this method constructs an instance of the `MCInst` class, which is the internal datastructure for machine instructions in LLVM-MC. For each of the remaining operands, it is checked whether their corresponding `@operand` grammar elements in the instructions grammar rule where assigned to a format field attribute or a field access function attribute. In the former case, the parsed value is adjusted by applying the field access function to transform the value to its field access view, in order for the assembler to have a consistent internal view of operands for all instructions. After this step the operand is added to the `MCInst` instance of the instruction. Additionally,

immediate operands are also checked to be within their valid value range as given by the available bits in the instruction format.

RecursiveDescentParser

As stated previously the `RecursiveDescentParsers` task is to parse instructions and create a list of the parsed operands. For this purpose OpenVADL generates a parser from the specifications in the assembly description `grammar` based on the lexical analysis provided by LLVM-MC. As recursive descent is used as parsing algorithm, the parsers internal structure closely reflects the structure of the specified `grammar`. For each non-terminal grammar rule a function of the same name is created. The functions body is determined by the grammar elements in the body of the rule. Grammar elements such as alternatives or optionals are modeled by if-constructs, repetition elements are translated to while loops and usage of terminal rules equal a call to the lexer expecting the terminal rules token. Usages of non-terminal rules are translated to calls to the functions generated for these rules. From the `MCAsmParsers` perspective the method generated for the `Statement` default grammar rule is the entry point into the `RecursiveDescentParser`. As it is an alternative over all grammar rules of AsmType `@instruction`, it allows parsing of all specified instructions. This cast to `@instruction` in the alternatives also alerts the `RecursiveDescentParser` to collect the attributes of the rule as operands and emit them to the `MCAsmParser`.

MCCodeEmitter

The `MCCodeEmitter` receives the instance of `MCInst` created by the `MCAsmParser` as input. First, it checks whether the passed instruction is a machine instruction or a pseudo instruction. As pseudo instructions have no direct binary representation, they need to be expanded to their equivalent machine instruction form. For each of the one or more resulting machine instructions the method `getBinaryCodeForInstr` is called. LLVM automatically creates its implementation in `GenMCCodeEmitter` from the definitions in the `InstrInfo.td` TableGen file. Listing 4.10 shows an example of the TableGen definition of the RISC-V ADDI instruction. It can be seen that it contains necessary information regarding the instructions format fields and their placement in the binary instruction word as defined in the VADL specification it is generated from (see 2.1).

The derived implementation of `getBinaryCodeForInstr` has different strategies for dealing with different types of format fields. For fields with constant values (e.g. `opcode` of Listing 4.10), values can simply be hardcoded into the implementation. For fields that encode register operands (e.g. `rd` of Listing 4.10) the implementation extracts the concrete operand value from the passed `MCInst` and calls into an implementation derived from the `RegisterInfo.td` TableGen file, which encodes registers to their binary representation.

Fields that hold immediate operands need more careful handling. Their corresponding operands can either be a numerical value or a symbol reference which needs to be resolved

```
def ADDI : Instruction
{
...
field bits<32> Inst;
...

bits<7> opcode = 0b0010011;
bits<3> funct3 = 0b000;
bits<64> immS;
bits<64> rs1;
bits<64> rd;

let Inst{31-20} = immS{11-0};
let Inst{19-15} = rs1{4-0};
let Inst{14-12} = funct3{2-0};
let Inst{11-7} = rd{4-0};
let Inst{6-0} = opcode{6-0};

...
}
```

Listing 4.10: TableGen definition of the RISC-V ADDI instruction

to an address at link time. In the first case it is to note that the operands passed in the `MCInst` instance are in the state of the field access view, but in machine code the field view is needed. To transform these operands, OpenVADL generates callback functions that apply the encoding function to an operands value. By naming the callback function in `InstrInfo.td`, the derived implementation of `getBinaryCodeForInstr` knows which callback function to call for a certain operand. In the case where an immediate operand is a symbol reference, the assembler needs to signal to the linker that this operand of this instruction needs to be replaced by the resolved address. This is achieved by using zero as placeholder value in the instruction word and emitting a relocation, which identifies the instructions location and operand. OpenVADL supports *absolute* and *relative* relocations. Absolute relocations need to be resolved to be the address of the referenced symbol. Relative relocations signal to the linker that it needs to determine the symbols address relative to the program counter of the instruction in question. The type of relocation to emit for a certain operand is chosen heuristically by analyzing the instructions behavior. In case the instruction is defined as adding the operand to the value of the program counter register, it is inferred that relocations for this operand are relative. For operands that do not meet this condition, an absolute relocation is emitted.

4.2.2 Linker

Similarly to LLVM-MC, the LLVM linker project LLD provides a large amount of target independent functionality. To create a linker for a new architecture, two key concerns need to be implemented. First, which relocations does the assembler produce? Second, how are resolved addresses integrated into the instruction word? OpenVADL generates these implementations based on definitions of the ISA section of a VADL specification. Both automatically generated relocations as discussed above and user defined relocations (see 4.1.1) are registered in the linker. Specifically, a mapping from each relocation to its kind (absolute or relative) is defined. This allows the target independent algorithm of LLD to resolve addresses in the expected fashion.

User defined relocations in the ISA are defined as functions that further transform resolved addresses. For each user defined relocation OpenVADL generates the corresponding transformation function to be applied by the linker. Resolved values of automatically generated relocations remain unchanged.

The final step in the linking process consists of adjusting the instruction word with the relocated value. For this OpenVADL generates the necessary encoding functions. An encoding function takes the unmodified instruction word and relocated operand value as input and writes the value into the operands bits in the instruction word.

CHAPTER 5

Evaluation

In this chapter we give measurements on the assembler and linker generation time, as well as a performance evaluation comparing assembler and linker generated by OpenVADL to LLVMs upstream implementation of the RISC-V 32 bit architecture.

All tests have been executed on a system with a Ryzen 9 7945 HX processor (16 cores), using 64 GB of DDR5 memory at 5600 MHz. Operating system on this machine is Ubuntu 24.04.2 LTS. Assemblers and linkers were generated based on LLVM version 19.1.7 and compared to the upstream implementations of the same LLVM version. Benchmarking measurements were taken with the Hyperfine[1] tool.

5.1 Tool Generation Performance

As discussed before in section 4.2, generating an assembler and linker with OpenVADL takes two distinct steps. First, parsing a VADL specification and generating sources to be used by the respective LLVM projects and second the compilation of these LLVM projects with the generated sources. Table 5.1 shows the average time of ten runs when generating an assembler and a linker for the RISC-V 32 architecture[2].

	Assembler	Linker
OpenVADL	195 ms	
LLVM Compilation	33.5 s	243.8 s

Table 5.1: Assembler and linker generation time taken for RISC-V 32 specification

The total time of about five minutes might seem to contradict OpenVADLs principle of rapid DSE, but it should be noted that these measurements are for the case of initial

[1] https://github.com/sharkdp/hyperfine

[2] VADL specification available in the OpenVADL repository https://github.com/OpenVADL/openvadl

compilation of the generated tools. In the case of repeated assembler and linker generation of an architecture, smart build tools and compilation caches used in LLVM greatly reduce the overall time it takes to generate executable artifacts. The reason for this is that the OpenVADL generated callbacks are concentrated in only a few source files, so on repeated execution only a small percentage of the total project sources have to be re-compiled.

For the same reason we expect similar overall results for other architectures / VADL specifications. Combined with the fact that the compilation step greatly outweighs the OpenVADL execution time, even large specifications should only marginally impact the overall execution time.

5.2 Assembling and Linking Performance

To evaluate the performance of the generated artifacts we again used the RISC-V 32 architecture VADL specification and compared it to the LLVM upstream implementation. As tests a subset of the RISC-V compliance suite RISC-V Architecture Test SIG[3] was used. This test suite focuses on architectural compliance, so test cases are not ”real world” programs, but rather focus on possible variations of a single instruction (e.g. test case `addi` contains thousands of `addi` instructions). Because of this the following results may not perfectly apply to ”real world” programs, but should still give a strong indication of the general assembler and linker performance.

Figure 5.1 shows the average performance of the OpenVADL generated assembler compared to the upstream LLVM implementation. While there are outliers, in most cases the generated assembler performs similar to the handwritten LLVM implementation as highlighted by the mean performance of 0.97 across all tests.

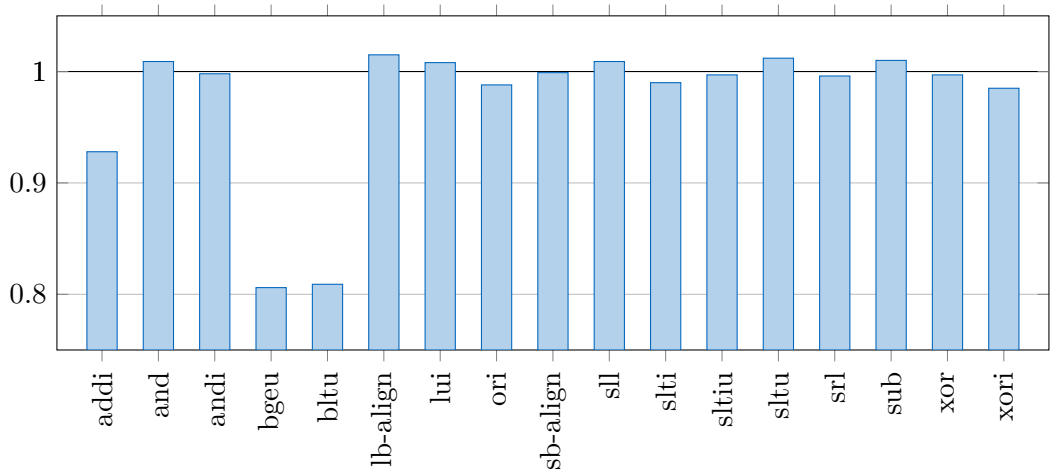

Figure 5.1: Performance of assembler relative to LLVM upstream (higher is better)

[3] https://github.com/riscv-non-isa/riscv-arch-test

To measure linker performance the files resulting of the assembling benchmark are linked with the respective OpenVADL and upstream LLD linkers. Figure 5.2 shows that like the assembler, the generated linker is similar in performance to the upstream LLVM version. There are again cases where the OpenVADL linker lacks behind, but the mean performance of 1.0003 across all tests suggests that the generated linker is on par with the upstream implementation.

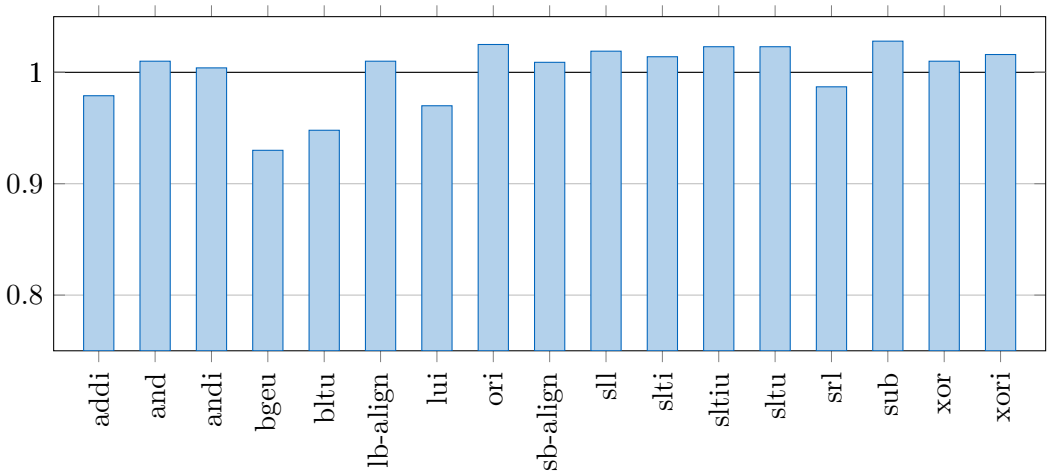

Figure 5.2: Performance of linker relative to LLVM upstream (higher is better)

CHAPTER 6

Future Work

Future work on the OpenVADL assembler and linker generators could focus on multiple dimensions. One aspect is the performance of the generated recursive descent parser in the assembler. The original VADL implementation employs an optimization that reduces comparisons when parsing alternatives over string grammar elements [Sch22]. A similar technique could be used in OpenVADL to increase the assembler's runtime performance.

Other efforts could focus on reducing the memory footprint of the recursive descent parser. As the parsing results of grammar elements are needed to build the instruction operand vector, the parsing result of each grammar element is held in a temporary variable. But since not each and every grammar element is relevant to the operand vector (e.g., purely syntactic characters like ","), some of these temporary variables are never read and are effectively unnecessary. Deeper analysis of the grammar could reduce the amount of (or even eliminate) unnecessary variables in the recursive descent parser.

Further work could also be done to expand the feature set of the assembly grammar. VLIW architectures feature the concept of instruction bundles. Support in the assembler generator would most likely require new grammar elements. Another idea is to extend the assembly grammar with the concept of inherited attributes. Currently, both attributes and local variables can only be synthesized. Being able to pass attributes as arguments to non-terminal rules would increase the expressiveness of the assembly grammar.

Finally, the most apparent extension to this work is the automatic derivation of grammar rules from the assembly printing definitions of the ISA section. This feature was available in the original VADL implementation and greatly reduces the specification effort. It also neatly aligns with VADL's aim of concise and non-redundant specifications. Due to several reasons, among other things the option to use arbitrary string functions in assembly printing definitions, this feature is highly non-trivial and did therefore not fit into the scope of this work. However, plans for its implementation as a follow-up to this work are underway.

CHAPTER 7

Conclusion

In this work, we extended OpenVADL to support the generation of assemblers and linkers. Using a similar approach to the original VADL implementation, we added an assembly description section to the OpenVADL frontend. Its grammar subsection was redesigned to feature EBNF-like syntax in an effort to increase the specification readability. A further improvement lies in the addition of semantic predicates to the set of available grammar elements. The resulting predicated-LL(1) grammar solves operand parsing issues rooted in the LL(1) limitation of the original VADL implementation.

To utilize synergies within LLVM the artifact generators were implemented in the context of OpenVADL's LCB. Based on the assembly description in its VIAM representation, a recursive descent instruction parser is generated. Parsed instructions are compared to instruction definitions of the ISA section and on successful matching passed on to the code emitter component. As the final step in the assembler, the code emitter is responsible for emitting the passed instruction's binary word.

The generated linker needs to apply user defined relocation functions and adjust instruction words with relocated operand values. To achieve this it derives the necessary information from instruction format and relocation definitions of the ISA section.

In our evaluation we observed that OpenVADL's assembler and linker generators are suitable for DSE in terms of artifact generation performance. In terms of execution performance, both generated assembler and linker for the RISC-V 32-bit architecture proved to be on par with their upstream LLVM equivalent.

Overview of Generative AI Tools Used

No AI tools were used during the creation of this work.

List of Figures

List of Tables

Acronyms

ABI application binary interface. 3, 5, 13, 18, 19

AD assembly description. 3

ADL architecture description language. 11, 12

DSE design space exploration. 1, 11, 12, 29, 35

EOL End-Of-Line. 18

GCB Generic Compiler Backend. 24

ISA instruction set architecture. 3, 5, 13, 15, 18–21, 28, 33, 35

LCB LLVM Compiler Backend. 8, 24, 35

LLD LLVM Linker. 8, 24, 28, 31

LLVM-MC LLVM Machine Code Playground. 8, 24–26, 28

PDL processor description language. 1

VADL Vienna Architecture Description Language. 1, 3, 5, 8, 9, 11, 13, 15, 23, 26, 28–30, 33, 35, 41

VIAM VADL Intermediate Architecture Model. 24, 35

Listings

Bibliography

[BCR05] A. Baldassin, P.C. Centoducatte, and S. Rigo. Extending the ArchC language for automatic generation of assemblers. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, pages 60–67, 2005. doi:10.1109/CAHPC.2005.25.

[BCR+08] Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz C. V. Santos, Max Schultz, and Olinto Furtado. An open-source binary utility generator. *ACM Trans. Des. Autom. Electron. Syst.*, 13(2), April 2008. doi:10.1145/1344418.1344423.

[CVDS06] Daniel C. Casarotto and Luiz C. V. Dos Santos. Automatic Link Editor Generation for Embedded CPU Cores. In *2006 IEEE North-East Workshop on Circuits and Systems*, pages 121–124, 2006. doi:10.1109/NEWCAS.2006.250893.

[FHH+25a] F. Freitag, L. Halder, B. Huber, B. Kasper, M. Nestler, K. Per, M. Raschhofer, A. Ripar, J. Zottele, and A. Krall. OpenVADL: An Open Source Implementation of the Vienna Architecture Description Language. In S. Tomforde, C. Krupitzer, S. Vialle, E. Suarez, and T. Pionteck, editors, *Architecture of Computing Systems: 38th International Conference, ARCS 2025, Kiel, Germany, April 22–24, 2025, Proceedings*, Lecture Notes in Computer Science. Springer Cham, 2025.

[FHH+25b] 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, and Andreas Krall. The Vienna Architecture Description Language, 2025. arXiv:2402.09087.

[Gon00] R.E. Gonzalez. Xtensa: a configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000. doi:10.1109/40.848473.

[HHD00] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration. *Design Automation for Embedded Systems*, 6(1):39–69, September 2000. doi:10.1023/A:1008937425064.

[HNP+01] A. Hoffmann, A. Nohl, S. Pees, G. Braun, and H. Meyr. Generating production quality software development tools using a machine description language. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 674–678, 2001. doi:10.1109/DATE.2001.915097.

[KL09] Per Karlström and Dake Liu. NoGAP: A Micro Architecture Construction Framework. In Koen Bertels, Nikitas Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 171–180, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-03138-0_18.

[KLAL10] Per Karlström, Sumathi Loganathan, Faisal Akhlaq, and Dake Liu. Automatic assembler generator for NoGap. In *6th Conference on Ph.D. Research in Microelectronics & Electronics*, pages 1–4, 2010.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[llv25a] LLVM Project. https://llvm.org, 2025. Accessed: 2025-07-25.

[llv25b] LLVM TableGen. https://llvm.org/docs/TableGen/, 2025. Accessed: 2025-07-25.

[MLW18] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The Compiler Generator Coco/R, 2018. URL: https://ssw.jku.at/Research/Projects/Coco/.

[OG98] S. Onder and R. Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*, pages 80–89, 1998. doi:10.1109/ICCL.1998.674159.

[PHZM99] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA-machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 933–938, 1999. doi:10.1109/DAC.1999.782231.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. doi:10.1002/spe.4380250705.

[PQD93] Terence Parr, R. Quong, and Henry Dietz. The Use of Predicates In LL(k) And LR(k) Parser Generators (Technical Summary). *ECE Technical Reports*, 01 1993.

[RABA04] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: a systemC-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73, 2004. doi:10.1109/SBAC-PAD.2004.8.

[Sch22] Tobias Schwarzinger. Flexible generation of low-level developer tools with VADL. Master's thesis, Technische Universität Wien, 2022. doi:10.34726/hss.2023.103246.

[SWZR14] Jia Qi Shen, Jun Wu, Zhi Feng Zhang, and Hao Qi Ren. Design and Implementation of Binary Utilities Generator. In *Machine Tool Technology, Mechatronics and Information Engineering*, volume 644 of *Applied Mechanics and Materials*, pages 3260–3265. Trans Tech Publications Ltd, 11 2014. doi:10.4028/www.scientific.net/AMM.644-650.3260.

[WLM03] Albrecht Wöß, Markus Löberbauer, and Hanspeter Mössenböck. LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages*, pages 192–201, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. URL: https://ssw.jku.at/Research/Papers/Woe03/WoeLoeMoe03.pdf.

[XL23] Xin Xiao and Zhong Liu. ISADL: An Instruction Set Architecture Description Language for VLIW. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 92–99, 2023. doi:10.1109/ICPADS60453.2023.00022.