TU WIEN Informatics

LLVM Compiler Generator in OpenVADL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Kevin Per, Bsc.
Matrikelnummer 11731660

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 27. November 2025

Kevin Per	Andreas Krall

TU WIEN Informatics

LLVM Compiler Generator in OpenVADL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Kevin Per, Bsc.
Registration Number 11731660

to the Faculty of Informatics

at the TU Wien

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

Vienna, November 27, 2025

Kevin Per	Andreas Krall

Erklärung zur Verfassung der Arbeit

Kevin Per, Bsc.

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, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. November 2025

Kevin Per

Acknowledgements

I wish to thank my parents and my sister, Grzegorz, Krystyna and Julia, for their mental and financial support throughout my academic journey.

I would also like to thank Prof. Andreas Krall for his support, guidance and excellent supervision during the course of this thesis.

Finally, I would like to thank the VADL team, especially Johannes and Benedikt, for their valuable feedback and collaboration.

Kurzfassung

Das Entwickeln von Prozessoren ist eine arbeitsintensive Aufgabe. Jede Änderung erfordert Änderungen der Toolchain. Gleichzeitig benötigt schnelle Entwicklung auch schnelle Entwicklungsiterationen. Die Vienna Architecture Description Language (VADL) ist eine Prozessorbeschreibungssprache, wo die Spezifikation als goldenes Modell dient. Die quelloffene Implementierung, OpenVADL, generiert mehrere Artifakte für das schnelle Entwickeln von Prozessoren aus der Spezifikation. Diese Arbeit präsentiert den Compiler Generator für einen LLVM-basierten C Compiler.

Die größte Herausforderung ist die Ableitung von Instruktionsselektionsmustern aus den Instruktionsverhaltensgraphen in einer VADL Spezifikation. Unser Ansatz analysiert und klassifiziert jede Instruktion anhand des Instruktionsverhaltens, um dann systematisch Muster für jede Klasse zu generieren.

Schließlich werden die generierten gegen öffentlich verfügbare Compiler für die Architekturen RV32IM, RV64IM und AArch64 verglichen.

Abstract

The development of processors is a labor-intensive task. Every design change requires the corresponding adaptation across the toolchain. At the same time, fast development requires fast iterations. The Vienna Architecture Description Language (VADL) is a processor description language in which the specification acts as golden model. Its open source implementation, OpenVADL, generates a variety of tools for rapid prototyping of processors. In this thesis, a compiler generator is presented that generates an LLVM-based C compiler from a VADL specification.

The biggest challenge is deriving the instruction selection patterns from the instruction behavior specified in the VADL specification. Our approach addresses this by analyzing and classifying instructions according to their behavior, and then systematically generating instruction selection patterns for each classification label.

Finally, we evaluated the generated compilers against the publicly available compilers for RV32IM, RV64IM and AArch64, respectively.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Structure of this Work 1

2 Background **3**
2.1 Compiler Backend 3
2.2 Processor Description Languages 17
2.3 Vienna Architecture Description Language 19
2.4 OpenVADL 24

3 Implementation **29**
3.1 Research Questions 29
3.2 Overview 30
3.3 Encoding Generation 31
3.4 Predicate Generation 32
3.5 Operand Detection 33
3.6 Instruction Classification 34
3.7 Lowering Machine Instructions 37
3.8 Lowering Pseudo Instructions 40
3.9 Lowering Registers 41
3.10 Relocations 42
3.11 Instruction Expansion 43
3.12 Constant Materialization and Register Adjustment 43
3.13 Conditional Branch and Compare-and-Jump Instructions 44

4 Problems **45**
4.1 Recursive Lowering 45
4.2 Frontend and Backend Type Mismatch 46
4.3 Truncation 46

4.4 Register index with different register files . . . 47
4.5 Unused operands for pseudo instructions . . . 47
4.6 Instruction Alignment . . . 48
4.7 Type Mismatches during Instruction Expansion . . . 48
4.8 Free Truncations and Extensions . . . 48
4.9 Application Binary Interface . . . 49

5 Evaluation **51**
5.1 RISC-V . . . 51
5.2 AArch64 . . . 53

6 Related Work **57**
6.1 LISA . . . 57
6.2 ArchC . . . 58
6.3 xADL . . . 58
6.4 Original VADL Compiler Generator . . . 59
6.5 VEGA . . . 60
6.6 Completeness and Translation Validation . . . 60

7 Future Work **63**
7.1 Missing concept for unmapped rules . . . 64
7.2 Multi Output Instructions . . . 64
7.3 Constant Sequences . . . 64

8 Conclusion **67**

Overview of Generative AI Tools Used **69**

List of Figures **71**

List of Tables **73**

List of Algorithms **73**

Acronyms **75**

Bibliography **77**

CHAPTER 1

Introduction

Moore's law predicted that the number of transistors on a chip would double every 18 months. While this has been true for many decades, modern engineering has struggled to keep up lately. New applications, such as artificial intelligence, have entered the stage and require new engineering paradigms to meet their power and performance demands. In recent years, computation has moved from homogeneous general-purpose processors to co-processors, that are tailored for specific applications. Artificial intelligence is now trained on hundreds of GPUs. While training of large language models is being done in large data centers, even consumer hardware added additional processors for acceleration. This heterogeneous paradigm, which offloads computation to additional processors, is not new but quite popular at the moment. General-purpose processors need a lot of power and chip area. However, as stated in [GM04, p2], flexibility and energy efficiency are competing goals. The word "flexibility" is understood as using a single processor for multiple applications.
By introducing a new Processor Description Language (PDL), we would like to ease the design flow of prototyping a new processor [Fre+]. This thesis introduces a compiler generator to target an Application Specific Integrated Processor (ASIP). Given a specification in the Vienna Architecture Description Language (VADL), presented in [Fre+], OpenVADL's compiler generator can generate a compiler.

1.1 Structure of this Work

Chapter 2 provides essential context on the operation of compiler backends and VADL. Chapter 3 discusses implementation details. Chapter 4 addresses key limitations of the current implementation, while Chapter 7 considers future directions and shows possible extensions. Chapter 5 discusses the performance of the generated compiler. Chapter 6 presents related PDLs, which inspired VADL. Finally, Chapter 8 presents a conclusion, final reflections and tying together findings and their implications.

CHAPTER 2

Background

This chapter describes the most important aspects of a compiler backend and introduces VADL.

2.1 Compiler Backend

A compiler is separated into multiple modules. While a modern compiler like LLVM has many components, there are three major modules: frontend, middleend and backend.

- The *frontend* handles the input language like for example C or C++. Programs are parsed, type checked and finally converted into an Intermediate Representation (IR) e.g. the LLVM Intermediate Representation (LLVM IR).
- The *middleend* applies *target-independent* transformations on the IR. These optimizations are independent of the programming language and the target-machine.
- The *backend* lowers the IR to assembly or machine code to produce an executable program for the hardware and applies optimizations based on the target Instruction Set Architecture (ISA), which defines the instructions supported by a processor.

Compiler backends can be distinguished into three essential phases: instruction selection, instruction scheduling, and register allocation. These three phases will be discussed in more detail in this chapter. The next section discusses different intermediate representations commonly used in compilers.

2.1.1 Intermediate Representations

A textual representation of a program would be too difficult to optimize. Compiler frontends create an IR with properties more suitable for optimizations. There are many

different kinds of IRs. A classification overview is presented in [Coo12, p162]. A program traverses multiple representations during the compilation until it is transformed to assembly or machine code. In practice, a compiler has multiple representations of a program because different compilation stages require different levels of abstraction and analysis.

Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree-structured representation of an input program. It is abstract because it omits irrelevant details, such as the code's formatting and comments. Therefore, it encapsulates a program's syntactic structure. A node represents a syntactic element and an edge is a structural relationship between two nodes. Internal nodes are statements or operations in the tree. While leaf nodes are variables or constants. The AST is usually the first IR in the compilation process.

Control Flow Graph

A Control Flow Graph (CFG) represents the order of execution as a graph. Each node represents a basic block. A directed edge in a CFG models the control transfer between different basic blocks. A basic block is a sequence of instructions where control flow can only enter at the beginning and can exit only at the end. The terminator is the last instruction of the block and determines which basic block is executed next. Common terminators are unconditional jumps, conditional jumps and return instructions. Some CFG representations can model a *fall-through* where no explicit terminator is required and the next successor is automatically executed next. A CFG represents all possible execution paths in a function [Coo12, p170].

Static Single Assignment Form

Static Single Assignment Form (SSA) is not an IR itself. It indicates that the IR's operations uphold certain guarantees [RB22]. In this representation, every variable has only one definition, and multiple definitions are joined by a phi function when control flow merges. A phi function is called a phi node when the IR is graph-based. A phi function's semantics is to copy the value depending on the edge from which the predecessor basic block originates. In practice, SSA has an unlimited number of *virtual registers*. Since only one definition is allowed, a virtual register represents a version of a variable. According to [Coo12, p194], this form has the benefit of simplifying lifetimes of values for optimizations.

Dataflow Graph

A Data Flow Graph (DFG) is a graph-based IR that depicts the data flow between nodes. A node represents an operation in a program, and a directed edge represents an operation's input. A drawback is that conditionals have to be modeled using select or phi nodes in a DFG, as conditionals require exclusive execution, with only one branch active at a time. On the other hand, parallel execution executes both branches independently

[WP94]. A select node is an operation that chooses one of multiple input values based on a boolean condition.

Program Dependence Graph

Another graph-based IR is the Program Dependence Graph (PDG). It combines both a control-dependence and a data-dependence graph into one representation [Kuc+81] [FOW87] [Muc06, p284]. Each node represents an operation in the program. The control-dependence subgraph shows whether a node A can directly affect whether a node B is executed. A data-dependence is a constraint between two statements A and B where the reordering of the nodes would result in an incorrect result [Muc06, p268]. There are different kinds of data-dependence constraints. If A writes a value which B reads then the constraint is called *flow dependence*. If A reads a value which B writes then the constraint is called *anti dependence*. If A and B set the same value then the constraint is called *output dependence*. If both A and B read a value from some variable then the constraint is called *input dependence*. The data-dependence constraints modeled as directed edges between two statements are summarized in a data-dependence graph.
The benefit of this IR is that the optimizing compiler can enable powerful optimization by inspecting the graph. The compiler can safely reorder instructions since the dataflow constraints are modeled in the graph. Additionally, the graph enables loop transformations like loop fusion or vectorization [FOW87].

Linear IRs

According to [Coo12, p175], "Linear IRs represent the program as an ordered series of operations". Cooper continues by defining that a one-address code models data through the stack or accumulator registers. A three-address code represents an operation with two source operands and a result. Three-address code is also called *quadruple code* because operation code, two source operands and a result are composed to a tuple of four components. A linear IR has two benefits. First, it is very close to a hardware representation since assembly code is also linear. Second, the IR is easily serializable and deserializable. This is an important property for debugging. On the other hand, linear IRs do not typically contain high-level constructs such as loops. This is a problem for advanced optimizations, since retaining metadata like the loop stride enables more effective optimization analyzes.
Typically, linear IRs are embedded in a CFG where each basic block has a linear sequence of instructions.

Hybrid IRs

Hybrid IRs are a mixture of multiple kinds. An example is Multi Level Intermediate Representation (MLIR) [Lat+21]. MLIR is structured into many different dialects. A dialect is an IR extension which defines a set of domain specific operations. Each dialect is also responsible for individual transformations and optimizations. Each dialect has its

own abstraction because some define high-level constructs, while others define low-level details. By progressively lowering the dialects, the IR becomes unified. Lowering is the process of transforming one dialect into another dialect. For most dialects the lowering ends with the LLVM dialect which can be translated to LLVM IR.

LLVM IR

The LLVM compiler frontend reads a source program and translates it into LLVM IR. The middleend applies target-independent optimizations, while the backend maps the IR into machine instructions. LLVM IR has two external representations: a bitcode and a human-readable textual format. The representation is composed of multiple modules, each serves as a compilation unit for the concurrent translation. LLVM IR is statically typed and is in SSA [Anoi].

The type system supports primitive types, labels, aggregate types, and functions. LLVM IR has attributes that support optimizations. For example, arithmetical instructions can indicate whether overflow or underflow can occur. Similarly, functions can also carry attributes, for example, to disable inlining. Floating point instructions can set fast-math flags that enable aggressive optimizations that may modify the result [Anoi]. However, one problem with LLVM IR is that it is very low-level. Because there is no explicit construct for loops, it is harder to apply loop optimizations. Typically, a loop's backedge is modeled with an unconditional jump to the basic block, which acts as the loop header. To address this, LLVM's middleend includes passes to recalculate loop bounds that are lost during translation [Anok] [Anoj]. Listing 2.1 shows a simple function that has three arguments $\%a$, $\%b$ and $\%c$. $\%a$, $\%b$ are integers and $\%c$ is a function pointer. The function has three basic blocks *entry*, $if.then$ and $if.end$. The two arguments a and b are compared with a `xor` operation and the result is stored into the virtual register %0. Based on the value of %0, the control flow continues with basic block $if.then$ or $if.end$. If both arguments are equal then the function pointer in virtual register $\%c$ is called. Afterwards, an unconditional jump jumps to the block $if.end$ which returns from the function. If both arguments are not equal then the control flow immediately continues with basic block $if.end$ and returns from the function.

2.1.2 Instruction Selection

In general, compilers translate source code into assembly or machine code, which is then executed by a target-machine. The process of mapping IR operations to machine instructions is called instruction selection. It is important that the selected instructions are semantically equivalent to the original program. While there are many possibilities to express semantic equivalence, instruction selectors compute a selection with the minimal cost.
For example, some architectures support a fused-multiply-add instruction. The instruction selector can choose whether it is profitable to use the fused instruction or separate additions and multiplications. Instruction selection can significantly influence the program's performance and size.

```
define void @bool_eq(
   i1 zeroext %a, i1 zeroext %b, ptr nocapture %c) nounwind {
entry:
  %0 = xor i1 %a, %b
  br i1 %0, label %if.end, label %if.then

if.then:
  call void %c() #1
  br label %if.end

if.end:
  ret void
}
```

Listing 2.1: LLVM IR example

The set of available machine instructions is defined in the ISA. The following subsections discuss various instruction selection strategies.

Macro Expansion

With this instruction selection technique, multiple templates are defined. A template can be the assignment of a variable or an operation performed on one or more operands, which can subsequently be expanded. The selector traverses the program's IR elementwise and emit the corresponding assembly code when a template is matched [Hjo16, p13]. An example is shown in Listing 2.2 from [Hjo16, p13]. The snippet shows C statements, where each statement is translated into a sequence of machine instructions. The selected machine instructions are written as comments. The example represents just a high-level idea.

```
   int a = 1;     // mv r1, 1
   int b = a + 4; // add r2, r1, 4
   p[4] = b;      // mv r3, @p
                  // add r4, r3, 16
                  // st r4, 0(r2)
```

Listing 2.2: Macro Expansion Example from [Hjo16, p13]

The macro expansion has limitations regarding the code quality and performance [DF84]. A peephole optimizer goes over the emitted machine instructions and tries to combine multiple instructions into one machine instruction. An IR is expanded into sequence of register transfer lists *RTL*. This representation is close to an assembly representation. The peephole optimizer's combiner then runs over the RTL and tries to combine them into a

larger RTL. Each target-machine's instruction also has a RTL. The combiner tries to merge the program's RTL expressions into one which corresponds to an instruction's RTL.

Equation 2.1 shows the example for a RTL in [Hjo16, p23]. The *add* instruction has two observable effects. The first is the result of an addition with an immediate value and the second observable effect is the write to a status flag. As also noted in [Hjo16, p23], the effect on the status flag is only relevant when a subsequent instruction reads the flag. If no other instruction reads the flag then it becomes unobservable. Hence, it can be removed which increases the chances to be combined into a larger RTL which might lead to performance gains.

$$RTL(add) = \left\{ \begin{array}{l} r_d = r_s + imm \\ Z = (r_s + imm) \iff 0 \end{array} \right\} \tag{2.1}$$

A popular compiler framework, GNU Compiler Collection (GCC), uses macro expansion together with a target-independent peephole optimizer. The GCC's low-level IR, known as *Register Transfer Language*, is a LISP-like language. A particular node is called *Register Transfer Language Expression* in the IR. GCC transforms a program into RTL and finally applies the *RTL Optimizer* to improve the generated code. GCC's instruction selector is a modified version of the Davidson-Fraser approach that was presented above [DF04] [Anod].

Tree Covering

Another approach for instruction selection is called tree covering. As already mentioned, some IRs can be in tree form. When both instruction patterns and IR are represented as trees, they can create a cover of the IR tree. Since multiple instruction patterns may cover the same IR tree, only patterns from the minimal cost cover should be considered. An optimal solution can be computed in linear time using bottom-up dynamic programming [Hjo16, p58].
In [AGT89], *Twig* was introduced as a tree-based notation for implementing code generators. First, the algorithm builds a top-down tree automaton from instruction patterns. Cost computation is based on Aho and Johnson's dynamic programming algorithm, which computes a program cover by selecting a set of pattern trees [Hjo16, p58]. That is because many covers overlap, and it is desired to get the cover with the minimal costs.

Restrictions. However, tree covering comes with multiple restrictions. First, dataflow is usually not represented as tree, but generally as Directed Acyclic Graph (DAG). A DAG is a graph with directed edges which has no cycles. Compared to a tree, a DAG may have multiple roots. Second, common subexpressions cannot be modeled [Hjo16, p76]. Unfortunately, tree covering is limited to tree patterns.

DAG Covering

Both the IR and the instruction selector may have different representations. There are three different cases to distinguish:

1. Section 2.2 discusses the case when the IR and the instruction selector are trees.
2. An IR can be a DAG and the instruction selector is a tree.
3. Another case is when IR and the instruction selector are both DAGs.

Unfortunately, the most interesting case is the third case and such a covering cannot be found optimally since it is NP-complete [Hjo16, p78]. In [Ert99] [TEK18], Ertl shows that an optimal selection can be achieved by extending tree parsers for IR DAGs, but the approach fails to produce optimal results in all cases.
In [Hjo16, p84], the process of "undagging" a DAG is described, where "undagging" refers to converting a DAG into a set of separate trees by splitting edges at points where common subexpressions occur. Afterwards, tree covering algorithms can be applied to each extracted tree. A common subexpression is easily detectable because two adjacent nodes have multiple edges between them.
Instruction selection can also be formulated as *Partioned Boolean Quadratic Problem* and solved efficiently for large instances [Ebn+08]. We refer to [Hjo16] where other DAG covering approaches are presented.

TableGen

LLVM uses a domain-specific language called TableGen to concisely define structured data. The two main data structures are records and classes. Records are concrete instances of *definitions*, while classes are blue prints for records. To create a record the keyword **`def`** and for a class the keyword **`class`** are used. Within a record, each entry is a key-value pair. Classes predefine key-value pairs which can be overwritten when a record inherits from the class. TableGen is used by different generators, where each definition represents a specific instance of an entity. TableGen is used for the following use cases: [Ano24]

1. Instruction definitions
2. Register definitions
3. Calling conventions
4. Command-line options
5. Diagnostic messages

TableGen is statically typed and supports bit, int, string, bit vector, list, direct cyclic graph as primitive types. The DAG type is mainly used for instruction selection definitions. TableGen also allows embedding C++ code at certain positions.

In Listing 2.3, the instruction `ADDI` defines multiple key-value pairs. The `OutputOperandList` sets the output operands. Likewise, `InputOperandList` defines the input operands. `ADDI` takes one register and an immediate as input operands. The `let Inst...` statements define the mapping from the operands to the instruction's binary representation. LLVM uses many target-independent passes that operate on target-machine instructions. These passes check the machine instruction flags and enable optimizations if applicable.

Listing 2.4 presents an instruction selection pattern in TableGen with two components. The first is the instruction selector that defines a tree pattern for the instruction selection. In this example, the root of the instruction selector graph is the *add* node, representing the LLVM IR addition operation in the Selection DAG Instruction Selection's Directed Acyclic Graph (SelectionDAG). The SelectionDAG is presented in Section 2.2. The selector's root has two children: an input register from register file X and the immediate value `$immS`. The second part specifies which machine instruction to emit. In Listing 2.4, the RISC-V `ADDI` instruction is emitted with operands matching those of the instruction selection pattern. The destination register is modeled implicitly because the `ADDI` record has an `OutOperandList` that defines the output as a register from the register file X.

Listing 2.3 shows an example instruction definition. In this case, `RV3264Base_ADDI_immSAsInt64` serves as the immediate operand for the `ADDI` instruction. This operand inherits from a `class`, which sets the `EncoderMethod` and `DecoderMethod`. The `EncoderMethod` is a C++ function that encodes the immediate operand into the binary representation. Similarly, the `DecoderMethod` is a C++ function that decodes the binary representation of an immediate operand. The `DecoderMethod` corresponds to the field access function defined in VADL.

```
def : Pat<(add X:$rs1, RV3264Base_ADDI_immSAsInt32:$immS),
    (ADDI X:$rs1, RV3264Base_ADDI_immSAsInt32:$immS)>;
```

Listing 2.4: Instruction Selection Pattern with TableGen (SDISel)

LLVM's Instruction Selectors

LLVM provides multiple instruction selectors that compiler backend engineers can use. The oldest selector is Selection DAG Instruction Selection (SDISel). It is limited to a selection based on basic blocks and single instructions. A faster selector is Fast Selection DAG Instruction Selection (FastISel) and the latest development is Global Selection Dag Instruction Selection (GlobalISel). The latest allows matching the entire function's scope. The following three sections explain the currently supported instruction selectors in LLVM. However, FastISel belongs to SDISel because it is embedded in it. It is possible to fallback from FastISel to SDISel in the middle of the basic block. This is not possible

```
// Defines a parameterized blueprint for the ADDI's immS
// immediate value.
class RV3264Base_ADDI_immS<ValueType ty> : Operand<ty>
{
  // C++ method to encode the value into the
  // binary representation.
  let EncoderMethod = "RV3264Base_ADDI_wrapper";
  // C++ method to decode the value from the
  // binary representation.
  let DecoderMethod = "RV3264Base_ADDI_immS_decode_wrapper";
}

// Concrete instance of ADDI's immS with a predicate function.
def RV3264Base_ADDI_immSAsInt64
    : RV3264Base_ADDI_immS<i64>
    , ImmLeaf<i64,
      [{ return RV3264Base_ADDI_immS_predicate(Imm); }]>;

// Concrete machine instruction for addition with register
// and immediate.
def ADDI : Instruction
{
// Output operands of the machine instruction.
let OutOperandList = ( outs X:$rd );
// Input operands of the machine instruction.
let InOperandList =
  ( ins X:$rs1, RV3264Base_ADDI_immSAsInt64:$immS );
// Encodings of the binary representation.
field bits<32> Inst;
bits<7> opcode = 0b0010011;
bits<3> funct3 = 0b000;
bits<64> immS;
bits<64> rs1;
bits<64> rd;
// Mappings of the instruction's operand into the binary
// representation.
let Inst{31-20} = immS{11-0};
let Inst{19-15} = rs1{4-0};
# ...
// Machine instruction flags that are used in the
// target-independent optimizations.
let isTerminator       = 0;
let isBranch           = 0;
# ...
// Hardware registers that are being read from by the instruction.
let Uses = [  ];
// Hardware registers that are overwritten by the instruction.
let Defs = [  ];
}
```

Listing 2.3: Instruction Definition TableGen for ADDI

between GlobalISel and SDISel. A fallback from GlobalISel restarts the selection from the beginning [Col25, p385].
According to [Col25, p385], instruction selection follows three phases. First, the translation of the LLVM IR into a separate IR. Second, the legalization of IR constructs that are not supported by the target-machine. For example, a compiler frontend might produce an LLVM IR which relies on multiplication, while the target-machine does not support it. These differences need to be handled by replacing LLVM IR instructions with legal operations. Finally, the selection phase handles the translation from the selector's IR to the target-specific machine IR of the compiler backend.

SelectionDag

As already mentioned above, an optimal DAG covering is NP-complete. The SDISel splits each basic block into separate DAGs and starts the following process: [Col25, p391] [Hjo16, p81]

1. **SelectionDAGBuilder.** Initially, the LLVM IR needs to be translated into selection nodes called Selection Directed Acyclic Graph Node (SDNode) in the SelectionDAG. It is also possible to define custom nodes which can be matched later. However, they have to be replaced in a later stage.

2. **DAGCombine.** LLVM provides many optimizations that can be done on the SDNodes. The *DAGCombiner* fuses multiple nodes together when it can find a more efficient representation. These optimizations are target-independent. But, the compiler backend can also apply multiple target-dependent optimizations to combine SDNodes. This DAGCombine pass is run before the pre-legalization. Therefore, it will have nodes with types and operations which are not supported by the target-machine.

3. **Legalize.** The legalizer converts an illegal IR into a legal IR. Hence, the DAG will only contain types and operations that are supported by the target-machine.

4. **DAGCombine.** The DAGCombiner is run again and applies optimizations on the DAG but after the legalization.

5. **Selection.** Finally, the instruction selection is invoked and the SDNodes are mapped to machine IR.

6. **Schedule.** The last step is the scheduling which linearizes the sequence to create a machine basic block. A machine basic block contains only machine instructions, as all operations have been replaced accordingly.

At LLVM's compile time, the instruction definitions and instruction selection patterns defined in TableGen are translated into C++ code. According to [Ben], the instruction selection patterns are compiled into some kind of "byte code". All of these are stored in the *MatcherTable*. The instruction selector executes the entries one at a time. If an opcode, predicate, or type does not match, it skips entries to match the next pattern.

If all entries are exhausted and no match is found, the compiler throws a selection error, and the compilation stops. According to [Hjo16, p81], this approach is a greedy IR-DAG-to-DAG rewriter. It is referred to as a rewriter because a match on the DAG IR is replaced by a machine instruction tree. Therefore, a tree pattern matching is used for instruction selection.

Upstream compiler backends include peephole passes for more complex patterns. Upstream compiler backends are officially available LLVM compilers. A *downstream* compiler backend uses LLVM but is not part of the main LLVM project. An example of a peephole pass in the upstream AArch64 compiler backend is the *AArch64LoadStoreOpt* pass. The AArch64 ISA supports load and store memory instructions that can pre- or post-increment a pointer. These instructions cannot be modeled by tree patterns because the patterns would have multiple roots. The peephole pass iterates over the machine instructions after the instruction selection and fuses multiple instructions into a single machine instruction.

Listing 2.5 shows the LLVM IR for a function that takes two integers, adds them, and returns them. The example shows multiple register spills to the stack that are optimized when the IR is compiled with higher optimization levels. Figure 2.1 represents the code in Listing 2.5 before the DAG combine. There are three different kinds of nodes in the graph: data nodes, scheduling nodes, and glue nodes. Data nodes represent operations and values that are computed, scheduling nodes represent the order in which computations occur, and glue nodes enforce that specific instructions must be scheduled immediately after each other. Since the SelectionDAG is a DFG, the black edges represent use-def dependencies. The blue-dashed edges represent scheduling dependencies between nodes that enforce an ordering between the source and target nodes. Finally, glue dependencies indicate that it is important that two instructions stay together and are marked with red edges. This requirement is stricter than scheduling dependencies. Nodes with a glue dependency must stay together [Ano25a].
The numbers on the top of some nodes indicate the operand of the operation. The node's bottom row represents the results. Nodes may return multiple values. The type $i32$ defines the resulting type as a 32-bit signed integer. The chain operand *ch* defines an ordering constraint between source and target node. *EntryToken* indicates the beginning of a basic block. The graph's root is the last instruction in the block. *CopyFromReg* and *CopyToReg* model data flow between basic blocks.

Fast Instruction Selection

The benefit of using FastISel is faster compilation time. Some SDISel patterns are automatically translated for FastISel. However, it comes with restrictions. [Col25, p464] lists advanced selection patterns that are not supported. Two unsupported patterns are special matching functions. The third, a *ComplexPattern* allows custom matching functions written in C++. In AArch64, the immediate operand of an addition can be shifted. In the upstream implementation, the addition's immediate operand is merged

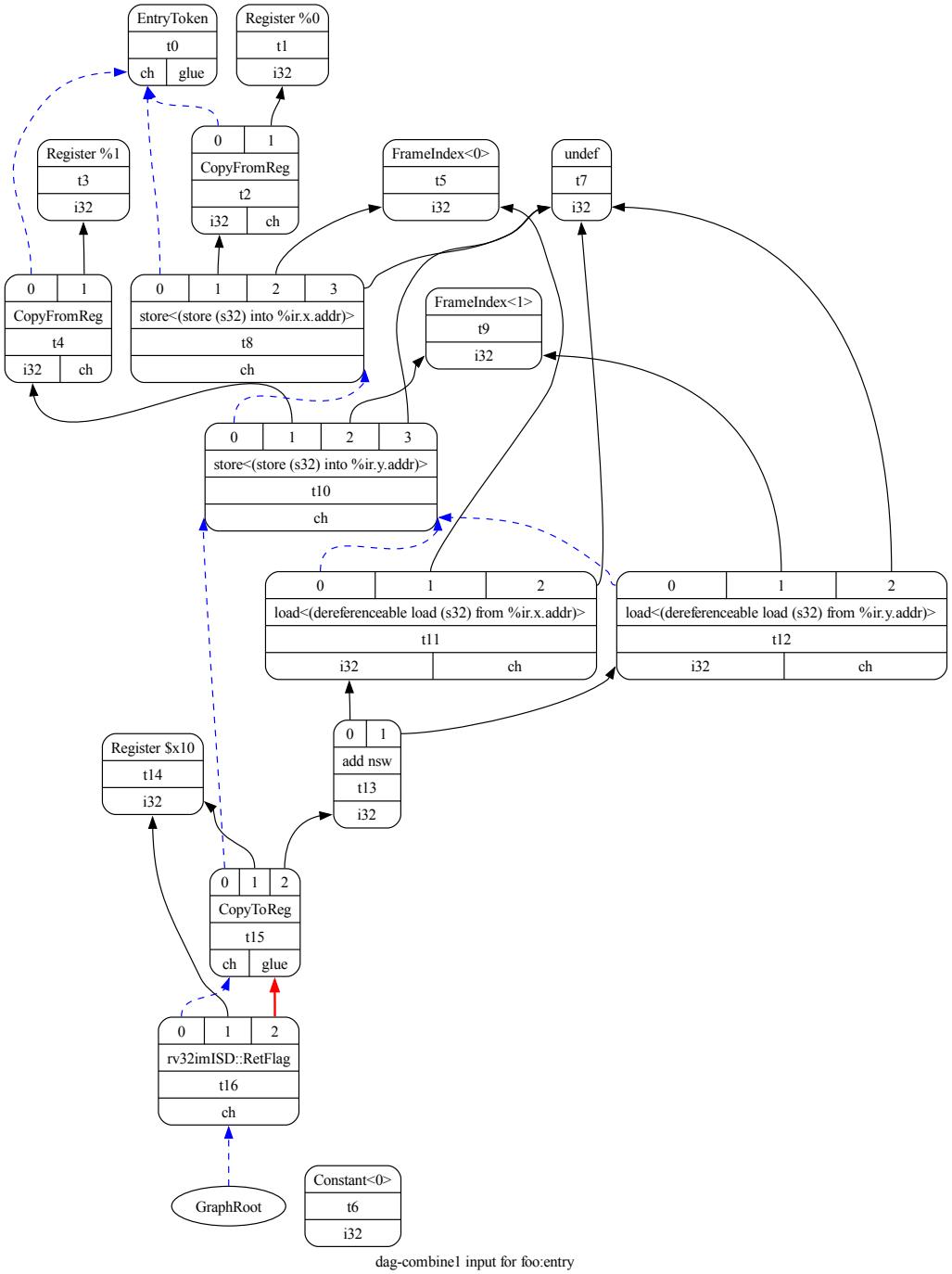

Figure 2.1: SelectionDAG for Listing 2.5

```
define dso_local i32 @foo(i32 noundef %x, i32 noundef %y) #0 {
entry:
  %x.addr = alloca i32, align 4
  %y.addr = alloca i32, align 4
  store i32 %x, ptr %x.addr, align 4
  store i32 %y, ptr %y.addr, align 4
  %0 = load i32, ptr %x.addr, align 4
  %1 = load i32, ptr %y.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

Listing 2.5: LLVM IR for a function with two inputs which are added and the result is returned

into a single *complex* operand, combining the value and the shift operands. The matching logic for handling the merged operand is manually done in C++. However, this is not automatically supported by FastISel unless it is implemented explicitly [Col25, p468].

Global Instruction Selection

The GlobalISel is an alternative instruction selection in LLVM that is considered to be a natural successor of SDISel. While SDISel operates at the basic block level, GlobalISel considers the entire function. A benefit of GlobalISel over SDISel is the *RegBankSelect* pass. While a register class groups registers by register file, a register bank is a functional group of registers. For instance, a register bank is the set of integer registers in the entire processor. A cross-register bank copy moves data from a general-purpose register to a vector register. It is preferable to avoid copying between register banks. Both SDISel and FastISel do not consider register banks [Col25, p454].
The following algorithm is documented by [Col25, p391] and [Anoa].

1. **IRTranslator.** GlobalISel has its own selection IR which is called Generic Machine Intermediate Representation (gMIR). This step translates the LLVM IR into gMIR [Anof].
2. **Legalizer.** It shapes the gMIR into a representation that is supported by the backend [Anog].
3. **RegBankSelect.** This pass constraints registers in gMIR to some register bank. This is useful when the backend supports multiple register files, then it avoids the copying between those [Anol].
4. **Select.** Finally, the gMIR is transformed into a representation which is target-specific. It starts at the bottom of the machine function and walks up to the definition [Anoe].

GlobalISel's selection pass is traversing over all the machine function's basic blocks in post-order. It then iterates from the end to the beginning of the function through all the gMIR instructions. The goal is to find a pattern which matches the gMIR. If the match was successful, it is mapped to a Machine Intermediate Representation (MIR). MIR is the low-level representation above the actual machine code. This greedy approach does not correspond to a DAG or tree covering, but is a one-to-one mapping. So then why is it called global? GlobalISel also includes a combiner which matches and fuses instructions together. Therefore, the scope on which the combiner operates is the entire function [Anoe].
The selection with the matcher table's byte-code remains the same as for SDISel.

2.1.3 Instruction Scheduling

According to [Hsu25], the machine scheduler has two objectives. It has to make the best use of the target machine's resources, while keeping the register pressure low. Especially for in-order micro architectures, instruction scheduling is important because it can improve the instruction-level-parallelism by reordering instructions [Col25, p491]. According to LLVM's documentation in *llvm/include/llvm/CodeGen/MachineScheduler.h*, the default scheduler is a list scheduler that updates the instruction stream, register pressure, and live intervals [Coo12, p634]. Subtargets can then choose a scheduling policy and a register-pressure-tracking policy. A scheduling policy is the direction of the scheduling. The top-down approach selects the next node with no successors. On the other hand, the bottom-up approach selects the next node with no predecessor. In other words, the top-down approach considers nodes from the beginning of the block first, while the bottom-up approach starts the scheduling from the last instruction of the basic block. Bidirectional is a mixture of both approaches, where the scheduler can schedule an instruction after all its successors or predecessors have been scheduled [Hsu25].

2.1.4 Register Allocation

LLVM uses LLVM IR as the intermediate representation of a program. This IR is in SSA [RB22]. LLVM IR assumes that there is an unlimited number of registers. Target machines, however, have only a restricted number of hardware registers. We also use the term *physical register* as a synonym. The register allocator assigns each virtual register a hardware register. Since there are more virtual registers than hardware registers, each hardware register is assigned to hold the values of multiple virtual registers. The allocator must compute the live ranges of each virtual register. A virtual register is considered dead when it has no usages and marks the corresponding hardware register as available. In many cases, there are more virtual registers alive than there are free hardware registers available. For these cases, the compiler stores the value temporarily in memory. This is called spilling. After the value is saved, the physical register can be reassigned. The previously stored value is not lost. When the old value is needed, it can be reloaded from memory. However, this is costly because accessing memory takes many cycles longer than accessing a register.

Register allocation maps virtual registers to hardware registers and determines which hardware registers are spilled. Spilling introduces new stores and loads, which influence instruction scheduling. Likewise, instruction scheduling can change register live ranges and influence register allocation.

There are multiple register allocation schemes, however, the two most prominent techniques are graph coloring and linear scan [Cha82] [PS99]. LLVM supports several register allocation algorithms [Anom]. The *fast* register allocator is the default for debug builds and does the register allocation per basic block. The *pbqb* (Partioned Boolean Quadratic Programming) allocator uses a PBQB solver to construct a solution [HKS03]. The *basic* allocator is useful for debugging issues. The *greedy* register allocator is the default allocator and is based on the previously mentioned *basic* allocator. Overall, it tries to minimize the cost of spill code. According to [Ole11], the greedy allocator tries to allocate large live ranges first. Small live ranges are allocated with the remaining set of registers. Live ranges of variables that do not fit are not spilled immediately. Instead, the live range is split. Thus, the range narrows and spilling becomes less likely.

2.2 Processor Description Languages

Design Space Exploration (DSE) for ASIPs is time-consuming and difficult, but necessary to accelerate a particular application. Additional machine instructions distinguish an ASIP from a general-purpose processor. However, custom hardware requires custom tooling, which slows iteration as each change requires updating the toolchain and simulators. This is a major challenge for ASIP designers and introduces a dependency between hardware and software. This dependency is called *hardware-software codesign problem*.

PDLs are designed to address the hardware-software codesign problem. A PDL is a language for describing a processor architecture. Thus, it is designed to generate a compiler, a simulator and a specification in a Hardware Description Language (HDL). The specification becomes the *source of truth*. Karuri et al. describe such a design flow in detail in [Kar+08]. However, their processor was intended to be only partially configurable. In [Hoh+04], Hohenauer et al. show how to model Infineon PP32 and ST200 microcontrollers. Scharwaechter et al. present a case study for IPSec encryption with ASIPs in [Sch+07]. Both articles used the PDL *LISA*.

PDLs are classified into two categories: specification-based or template-based [KL11, p8, p38]. A specification-based PDL is a high-level language that abstracts low-level details away, allowing only a limited degree of freedom. While for template-based specifications, Karuri and Leupers create two categories for customization. First, processor customization is achieved by combining existing features. Second, a PDL allows the specifier to add additional features.

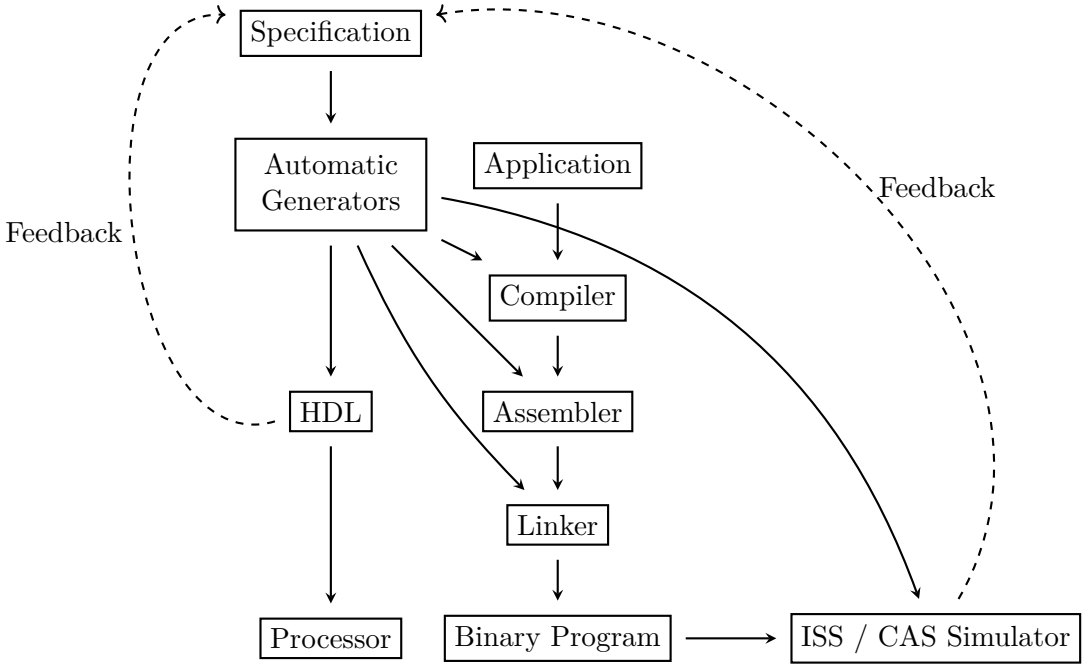

Figure 2.2: Specification based ASIP design flow from [KL11, p40]

Figure 2.2 shows an overview of a specification-based design flow for ASIPs from [KL11, p40]. The specification is used as an input for automatic generators. *HDL* specification, *Compiler*, *Assembler*, *Linker*, *Instruction Set Simulator* and *Cycle Accurate Simulator* were generated by automatic generators. The graph depicts two design flows.

1. The first design flow starts with the specification and ends with the specification in a HDL. The specification in the HDL can be synthesized into a processor. But during the DSE, feedback from the synthesis is incorporated into the processor specification.

2. The second design flow begins with the specification and generates the compiler, assembler, and linker. The toolchain's output is an executable, which is the input for the simulators. The feedback is incorporated into the specification based on the simulation results.

This process highlights the DSE because the specification is iteratively updated until the design meets specific constraints and exploration is complete. For embedded computing, energy efficiency, cost and size are key constraints that are opposed to each other and require trade-offs.

```
instruction set architecture RV32IM = {
    using Byte    = Bits<8>
    using Index   = Bits<5>
    using Address = Bits<5>

    program counter PC: Address
    register X : Index -> Bits<32>

    // Formats and Instructions are added here
}
```

Listing 2.6: RISC-V's ISA in VADL

2.3 Vienna Architecture Description Language

VADL is a language for describing processor architectures [Fre+]. It is structured into multiple sections to avoid *leaky abstractions*. An abstraction is leaky when low-level details have to be considered in high-level designs. The writer specifies a specification with separate ISA, Application Binary Interface (ABI), and Micro Architecture (MIA) sections. VADL was implemented by two projects. The latest and only active development is OpenVADL. Based on the given specification, OpenVADL generates multiple *artifacts*. An artifact is the compiler toolchain, Instruction Set Simulator (ISS), or a synthesizable processor in a HDL. The following sections present an overview of VADL.

2.3.1 Instruction Set Architecture

An ISA defines the supported machine instructions by the specified ASIP. An example of such a specification is listed in Listing 2.6. The keyword **`instruction set architecture`** defines a new ISA. ISAs are extensible, which is indicated by the **`extending`** keyword. The keyword **`using`** defines a type alias. The **`program counter`** defines a special purpose register that holds the memory address of the next instruction to be executed. The keyword **`register`** introduces a new register file called *X*. In this specification, the *X* register file consists of multiple registers, each addressed by a unique 5-bit index, and each register can store a 32-bit value.
An instruction has a type that is called *format*, and a format is created with the keyword **`format`**, as shown in Listing 2.7. The format defines an ordered list of fields, each with one or more bits. At the end of the format, a field access function, `immS = imm as SInt<32>`, is provided. This field access function serves as a separation between the instruction's behavior and its decoding. The instruction's behavior is explained in Section 2.4.1. By moving the decoding computation to the field access function, the reader can assume that the behavior operates on the decoded values, which increases readability. Applying the *immS* function in the instruction's behavior is exactly the same as reading the *imm* field, but with a sign extension.

```
format Itype : Inst =
{ imm    : Bits<12>        // [31..20] 12 bit immediate value
, rs1    : Index           // [19..15] source register index
, funct3 : Bits<3>         // [14..12] 3 bit function code
, rd     : Index           // [11..7]  destination register
, opcode : Bits<7>         // [6..0]   7 bit operation code
, immS   = imm as SInt<32> // sign extended immediate value
}
```

Listing 2.7: RISC-V's Itype instruction format in VADL

```
instruction ADDI : Itype =
    X(rd) := (X(rs1) as SInt + immS as SInt) as Regs
encoding ADDI =
{ opcode = 0b001'0011
, funct3 = 0b000
}
assembly ADDI = (mnemonic, " ", register(rd), ",", register(rs1),
",", sdec(imm as SInt<12>))
```

Listing 2.8: RISC-V's ADDI instruction in VADL

Figure 2.8 shows the *ADDI* instruction, which has the *Itype* format. The instruction has three components. The first is defined with the keyword **instruction** and implements the behavior. Next is the **encoding** that sets the bit fields for the `opcode` and `funct3`. The remaining fields `rs1`, `rd`, and `imm` are instruction operands and have no fixed bit representation. The third is the **assembly** keyword, which specifies how the compiler has to print the instruction in assembly. The function **register** maps a numeric value to a register file, e.g., *X10*. The other function **sdec** prints the value of the encoded field `imm` as a signed decimal number. **udec** can be used for unsigned decimal values and **hex** for signed hexadecimal representation.
Listing 2.9 shows an *encoding function* and a *predicate function*. An encoding function is the inverse function of the field access function. The terms *decode function* and *field access function* can be used interchangeably. The predicate function is a boolean function that determines whether a value is within range during instruction selection. In this particular case, *immS* must be in the range *-2048* and *2047* to be selectable. Otherwise, the instruction selector has to select another instruction. Why was it not necessary to define these functions in Listing 2.7? We refer the reader to Section 3.3 for the documentation on the function's generation. But the short answer is that the compiler generator can automatically derive encoding and predicate functions from field access functions of a certain form.

```
format Itype : Inst =
{ imm    : Bits<12>        // [31..20] 12 bit immediate value
, rs1    : Index           // [19..15] source register index
, funct3 : Bits<3>         // [14..12] 3 bit function code
, rd     : Index           // [11..7]  destination register
, opcode : Bits<7>         // [6..0]   7 bit operation code
, immS   = imm as SInt<32> // sign extended immediate value
, imm   := immS(11..0)     // encoding function
, immS  :- immS >= -2048 && immS <= 2047 // predicate function
}
```

Listing 2.9: RISC-V's Itype instruction full format in VADL

Registers

Listing 2.10 shows the declaration of registers in the AArch64 specification in VADL. The main register file is *S*. It has the type `Index -> Bits<64>`. The type is a function with 5 bits as an argument and returns 64 bit values. Therefore, the number of registers is 32. The *X* register file is an alias because it was created with the **`alias register`** keyword. The *W* register file is also an alias of *X*, but the type indicates that each register in *W* references only the lower 32 bits of a register in *X*.
The annotation `[zero : X(31)]` declares that *X(31)* is the zero register. The annotation **`overwrite source`** defines the zero extension of a subregister from *W* during writes. An alias has to be defined in AArch64 because it is possible to address the 32 least significant bits from a register in register file *S*.
Both *LR* and *SP* are aliases for a concrete register in *S*. Additionally, we define the status registers *NZCV_N*, *NZCV_Z*, *NZCV_C*, *NZCV_V*.
The importance of aliases becomes clear for two reasons. First, they can be constrained. For example, in Listing 2.10, reading from registers *S(31)* and *X(31)* return different values because the alias *X* has the zero annotation. The second reason is the definition of subregister relationships, explained in Section 3.9.

Relocations

Generally, program compilation has two stages. Each source code file is compiled into an object file. Afterwards, object files are linked to create a single executable. This process creates challenges for jumps and global variables. Jump instructions change the Program Counter (PC) to set the address of the next instruction. However, the function's address in the final executable is not determined at compile-time yet. Assemblers use placeholders in object files. The linker updates these placeholders because the addresses are known at link-time. These placeholders are called relocations.

The keyword **`relocation`** creates a custom relocation in VADL, as shown in

```
register        S : Index -> Bits<64>
[zero : X(31)]
alias register  X = S
[zero : W(31)]
[overwrite source: zero]
alias register W = X(*)(31..0)
alias register SP : Address = S(31)
alias register LR : Address = S(30)

[negative status register]
register NZCV_N : Bits<1>              // Negative
[zero status register]
register NZCV_Z : Bits<1>              // Zero
[carry status register]
register NZCV_C : Bits<1>              // Carry
[overflow status register]
register NZCV_V : Bits<1>              // Overflow
```

Listing 2.10: Register definition for AArch64 in VADL

Listing 2.11. The **syntax** annotation declares how the relocation is printed in the assembly representation. Brackets mark placeholders, which the compiler later replaces: the first for the relocation name, the second for the label name.

VADL supports different types of relocations that are differentiated by their annotation. For example, **relative** creates a relative relocation, while **global offset** creates a global offset relocation. Both `hi` and `lo` are absolute relocations. With absolute relocations, the linker directly updates the placeholder with the computed value. For relative relocations, the PC is subtracted from the computed value. Additionally, `got_pcrel_hi` applies to relocations involving the global offset table. The global offset table is necessary for compilation with position-independent code. VADL supports relocations for only one value.

2.3.2 Application Binary Interface

According to the Swift Stability Manifesto [Inc]: "[...] ABI is Application Binary Interface, or the specification to which independently compiled binary entities must conform to be linked together and executed. These binary entities must agree on many low level details: how to call functions, how their data is represented in memory, and even where their metadata is and how to access it."
To conclude, an ABI is a contract to ensure compatibility of binary entities between different toolchains.

```
[syntax : "%{}({})"]
relocation hi( symbol : Bits<32> ) -> UInt<20> =
    ( ( symbol + 0x800 as Bits<32> ) >> 12 ) as UInt<20>

[syntax : "%{}({})"]
relocation lo( symbol : Bits<32> ) -> SInt<12> =
    symbol as SInt<12>

[relative]
[syntax : "%{}({})"]
relocation pcrel_hi( symbol : Bits<32> ) -> UInt<20> =
    ( ( symbol + 0x800 as Bits<32> ) >> 12 ) as UInt<20>

[relative]
[syntax : "%{}({})"]
relocation pcrel_lo( symbol : Bits<32> ) -> SInt<12> =
    (symbol + 4) as SInt<12>

[global offset]
[syntax : "%{}({})"]
relocation got_pcrel_hi( symbol : Bits<32> ) -> UInt<20> =
    ( ( symbol + 0x800 as Bits<32> ) >> 12 ) as UInt<20>
```

Listing 2.11: Relocations for RISC-V in VADL

Listing 2.12 shows an ABI example for the ISA RV32IM. An ABI is created using the keyword **application binary interface**. **size_t type** defines *size_t* type in C, and for RISC-V, it must be unsigned. **alias register** sets names for assembly printing. Without an alias, it's unclear how to print a register from a register file. The different registers **stack pointer**, **return address**, **global pointer**, **frame pointer**, and **thread pointer** declare reserved registers. These are excluded from register allocation. **special return instruction** and **special call instruction** set instructions for function calls and returns.
special local address load instruction, **special global address load instruction**, and **special absolute address load instruction** load addresses. Multiple instructions allow the distinction between position-independent and non-position-independent code.
The **return value** defines registers for return values from functions. **function argument** specifies registers for passing values to function calls.
caller saved and **callee saved** define the calling conventions.

Listing 2.12 also contains three **constant sequence** and one **register adjustment sequence**. A constant sequence is an internal pseudo instruction for the constant

materialization. A pseudo instruction expands into several machine instructions, but it improves the readability in the assembly. Internal pseudo instructions are used by the compiler and are not printed in the assembly. Constant materialization builds a constant using machine instructions. The three sequences differ by type and size. It is more efficient to use one *ADDI* instruction if the immediate fits in its encoded field. Unlike a **`constant sequence`**, a **`register adjustment sequence`** takes an extra register argument. It adjusts a value instead of overwriting it. Register adjustment sequences are useful for stack adjustments, as a program must not overwrite the original stack pointer.

2.4 OpenVADL

OpenVADL is the main implementation of VADL [Fre+26] [Fre+]. The previous implementation is called Original VADL. However, it was an internally developed project and is not publicly available. [Fre+] describes the differences between the original implementation and OpenVADL in more detail. OpenVADL is its natural successor. A major difference between the two is the design of the IR. In addition, all generators were reimplemented from scratch. Previously, the ISS was a self-contained project, but it is now implemented on top of QEMU [Bel05]. The original and OpenVADL's compiler generator are both based on LLVM. However, the reimplemented generator is more advanced and supports higher optimization levels.
Figure 2.3 shows an overview of the architecture. Yellow boxes represent generators. The specification is parsed by the frontend, which expands all macros and creates an AST. Afterwards, the AST gets transformed into the VADL Intermediate Architecture Model (VIAM), OpenVADL's IR. All generators depend on the VIAM and modify its graph as needed. The generated compiler can be built without knowledge of the microarchitecture. The compiler generator is split into two components: Generic Compiler Backend Generator (GCB) and LLVM Compiler Backend Generator (LCB). The initial idea was to support multiple compilers with OpenVADL. We tried to move common functionality into a generic and reusable component. Nonetheless, LLVM is currently the only compiler supported by OpenVADL. The compiler generator is called LCB and sits on top of GCB. In reality, the GCB is a thin layer and design decisions were heavily influenced by LLVM.

2.4.1 VADL Intermediate Architecture Model

Our observation was that multiple generators require different representations. However, all the generators rely on control flow and dataflow. The VIAM serves as IR in OpenVADL where the instruction's behavior is modeled as a graph. Each generator tailors the IR to its needs by replacing or extending nodes in the graph. The compiler generator operates mostly on the instruction's behavior graph. The VIAM's instruction behavior graphs combine both control flow and dataflow into one multi-graph. Each behavior graph contains a start and an end node. The end node contains a list of side effects. Each side effect is a node in itself. For example, writing a register file is a side-effect node that has two inputs. One input is the destination index in the register file, where the write is performed. The

```
application binary interface ABI for RV32IM = {
   size_t type = unsigned int
   pointer align = 32

   alias register zero = X(0)
   alias register ra = X(1)
   alias register sp = X(2)
   alias register gp = X(3)
   alias register tp = X(4)
   // ...

   stack pointer = sp
   return address = ra
   global pointer = gp
   frame pointer = fp
   thread pointer = tp

   special return instruction = RET
   special call instruction = CALL
   special local address load instruction = LLA
   special global address load instruction = LGA_32
   special absolute address load instruction = LA

   return value = a{0..1}
   function argument = a{0..7}

   // ra is callee saved because it is used as
   // normal register and has to be restored.
   caller saved = [ a{0..7}, t{0..6} ]
   callee saved = [ ra, s{0..11} ]

  constant sequence( rd : Bits<5>, val : SInt<32> ) =
  {
     LUI { rd = rd, imm = hi( val ) }
     ADDI { rd = rd, rs1 = rd, imm = lo( val ) }
  }

  constant sequence( rd : Bits<5>, val : UInt<32> ) =
  {
     LUI { rd = rd, imm = hi( val ) }
     ADDI { rd = rd, rs1 = rd, imm = lo( val ) }
  }

  constant sequence( rd : Bits<5>, imm : SInt<12> ) =
  {
     ADDI{ rd = rd, rs1 = 0, imm = imm }
  }

  register adjustment sequence
    ( rd : Bits<5>, rs1: Bits<5>, imm : SInt<12> ) =
  {
     ADDI{ rd = rd, rs1 = rs1, imm = imm }
  }
}
```

Listing 2.12: ABI section in VADL for RV32IM

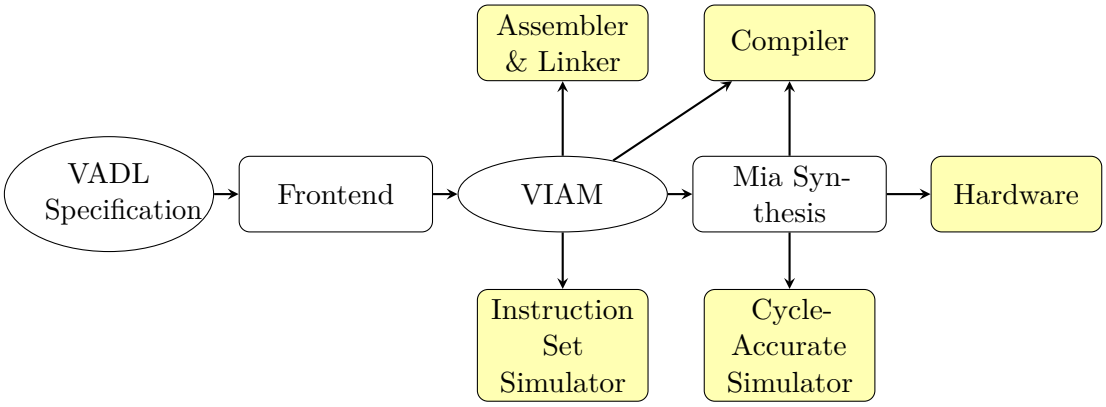

Figure 2.3: Overview of OpenVADL Compiler Architecture from [Fre+]. Yellow boxes represent generators.

other input is the value that is written. Reading from a register file and reading from a concrete register are modeled as single node that references the underlying resource. Dataflow is implicitly modeled in SSA form due to the semantics of let statements. All reads must precede all writes, and all writes are executed at the end of the block [Fre+].

Figure 2.4 shows an example of a RISC-V *ADD* instruction from [Fre+]. The *write<X>* is the side effect of the instruction, which has two inputs. One input is the value which is written, namely the *add: Bits<64>*. The second input is the node *field<rd>*, which specifies where the value is written. The addition node is itself integrated as an operation in VADL. They are also called *Builtins*. This thesis uses the terms *builtin* and *operation* synonymously. The builtin set is predefined and cannot be extended by the specifier. Each generator can modify this representation by adding, removing, or replacing nodes.

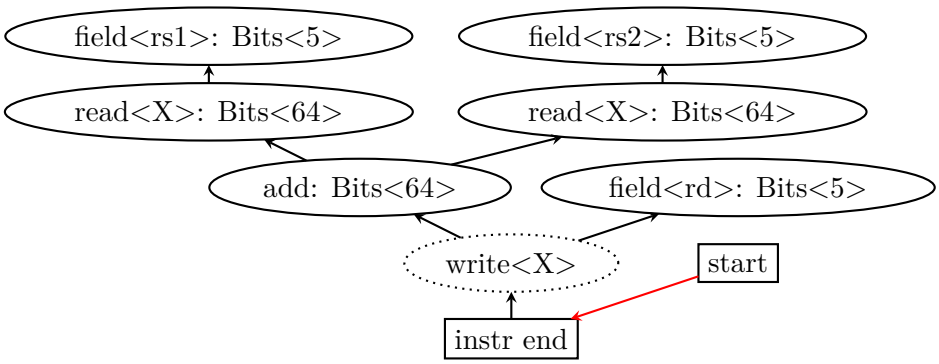

Figure 2.4: VIAM behavior graph of the RISC-V ADD instruction from [Fre+]

The VIAM has multiple types of nodes, such as:

- *BuiltinNode*. This node represents an operation like addition or a shift in the graph.
- *ReadRegTensorNode* or *WriteRegTensorNode*. This node models a read from or write to a register or a register file. It contains a reference to the underlying resource.
- *ReadArtificalResourceNode* or *WriteArtificialResourceNode*. An artifical resource is an alias to a register or register file.
- *ReadMemoryNode* or *WriteMemoryNode*. This node models the access to memory.
- *SelectNode*. A *SelectNode* is a conditional node that represents the selection of one of multiple input values based on a boolean condition input.
- *IfNode*. A *IfNode* is control-flow node that models the execution based on a boolean condition.
- *FieldRefNode*. This node represents the access to a field in a format.
- *FieldAccessRefNode*. This node represents the call to a field access function, defined in the format.
- *SideEffectNode*. The *SideEffectNode* is a term for a group of nodes that write a resource in the VIAM.

2.4.2 Semantic Gap

Generally, compilation of programs begins by translating source code into an IR, which is then translated into machine instructions. Figure 2.5 shows a compilation flow overview. The term *semantics* refers to the meaning of operations at each stage. A programming language first defines frontend semantics for operations such as addition. Next, LLVM IR provides its own IR semantics for this operation. Finally, machine instructions have specific semantics as well. As described in Section 2.1.2, instruction selectors match operations on the SelectionDAG or gMIR which are mapped from LLVM IR. However, the compiler generator must generate instruction selection patterns from the machine instruction behavior graphs specified in a VADL specification.

The *semantic gap* refers to the differences or mismatches between different semantics. The *Machine Instruction Semantic Gap* is the difference between what an instruction is supposed to do (semantics) and what the hardware actually executes (behavior). The *IR Semantic Gap* exists because compilers allow *undefined behavior*. Instructions like division have corner cases, such as division by zero. Undefined behavior allows the compiler to ignore corner cases [Anoh]. However, the machine instructions implement behavior by raising an exception or computing an invalid value. When generating instruction selection patterns, the compiler generator must bridge the gap between undefined and defined behavior, which we refer to as *IR Semantic Gap*.

Additionally, instruction behavior is specified with VADL operations and there is no one-to-one mapping between LLVM IR instructions and VADL operations. For instance, VADL includes an operation for addition that sets the status flags, whereas LLVM IR does not. In this work, we use the term *semantic gap* to summarize the *Machine Instruction Semantic Gap* and the *IR Semantic Gap*.

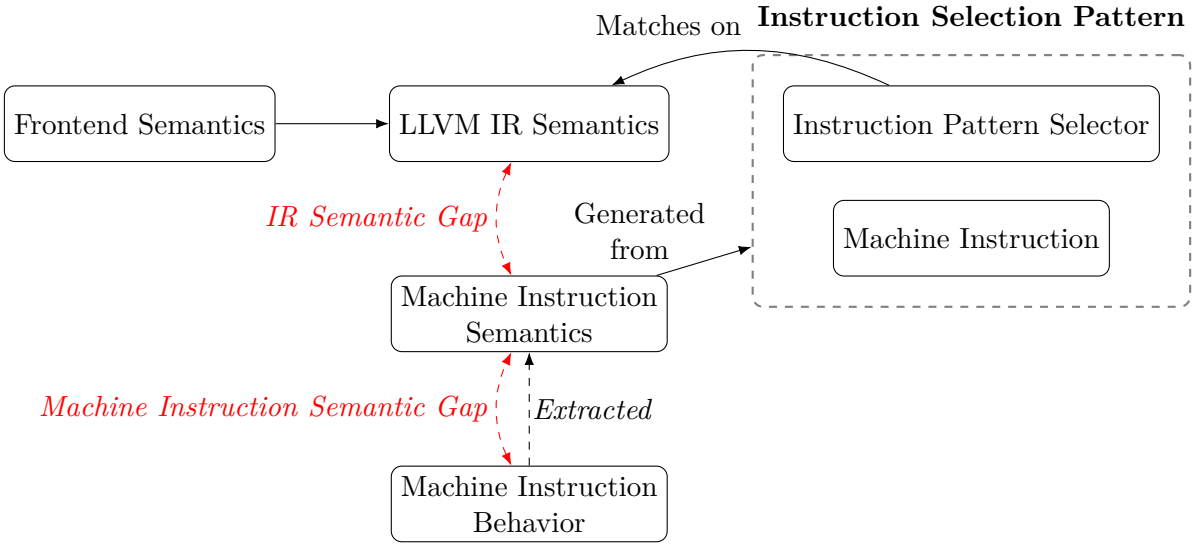

Figure 2.5: Semantic Gap in OpenVADL

CHAPTER 3

Implementation

The following sections discuss the implementation details of the automatic generation of a compiler, which is the main contribution of this thesis.

3.1 Research Questions

RQ1: *How can conditional branch instructions be automatically derived from a processor description language?*

Section 3.6 presents a classification algorithm that defines static properties for the behavior graphs of machine instructions in the VIAM. The algorithm assigns a classification label to each matched machine instruction when certain properties are fulfilled. For example, RISC-V has a single instruction that compares two registers and adjusts the PC based on the comparison. By contrast, AArch64 uses two separate instructions. First, a compare instruction compares two registers and sets the CPU flags depending on the result. The second instruction is a jump that interprets the CPU flags and updates the PC. These architectural differences require checking if a combined compare-and-jump machine instruction exists. If the VADL specification does not specify such an instruction, the compiler generator has to generate a compiler that chains together the individual compare and jump machine instructions to ensure correctness. Section 3.13 describes the implementation algorithm in more detail.

RQ2: *What is the semantic gap between instruction pattern selectors and instruction semantics?*

We refer the reader to Section 2.4.2 for a definition of the *semantic gap*.

RQ3: *What is the performance difference between VADL's clang and upstream*

clang for RV32IM, RV64IM and AArch64?

We evaluated RV32IM and RV64IM with the Embench suite [Anob]. For RV32IM, a program executed 29.73% more instructions than the clang upstream compiler on average for 21 of 22 benchmarks. For RV64IM, a program executed 43.83% more instructions than the clang upstream compiler on average for 20 of 22 benchmarks. All benchmarks were compiled with the highest optimization level *-O3*. We refer the reader to Section 5.1 for more details.

The generated AArch64 compiler does not support all the required functionality to run the Embench suite. We can compile and execute programs that use arithmetic, comparison and logical instructions. However, global variables, jump tables and indirect jumps are not supported yet. We refer the reader to Section 5.2 for more details.

3.2 Overview

The main contribution of this thesis is a compiler generator that automatically produces a compiler from a given VADL specification. OpenVADL is currently the only active implementation that supports VADL. OpenVADL reads the specification and transforms it into the VIAM, which is its IR. The project's architecture supports multiple generators, each producing an artifact that enables ASIP designers to perform DSE.

Figure 3.1 shows an overview of the compiler generator's architecture, which is split into two components. The GCB is responsible for the generic compiler generator functionality that is shared across multiple compilers and can be modularized. OpenVADL's distant goal is to generate multiple compilers, such as the GCC or a Java Just-In-Time (JIT) compiler. In contrast, the LCB's task is to generate a LLVM compiler.

The first task, in Figure 3.1, is the generation of encoding and predicate functions, which are described in Sections 3.3 and 3.4, respectively. Followed by the operand detection, which is described in Section 3.5. To conclude GCB's subgraph flow, the final step is the removal of branches that are edge cases, which is described in Section 3.6.1.

After these steps, the LCB analyzes the behavior of instructions, referred to as *instruction classification*. The classification algorithm assigns a classification label to each machine instruction. This label is used to generate instruction selection patterns, which are presented in Section 3.7.

Finally, the last step is to emit the rendered C++ and TableGen files. Each file has its own pass and is depicted as *File Emitter* in Figure 3.1. The generator's output is a set of files that overwrite an existing LLVM project. Afterwards, LLVM is compiled which results in an executable compiler called *clang*, which uses the VADL specification name as the target triple and acts as a cross-compiler for the specified

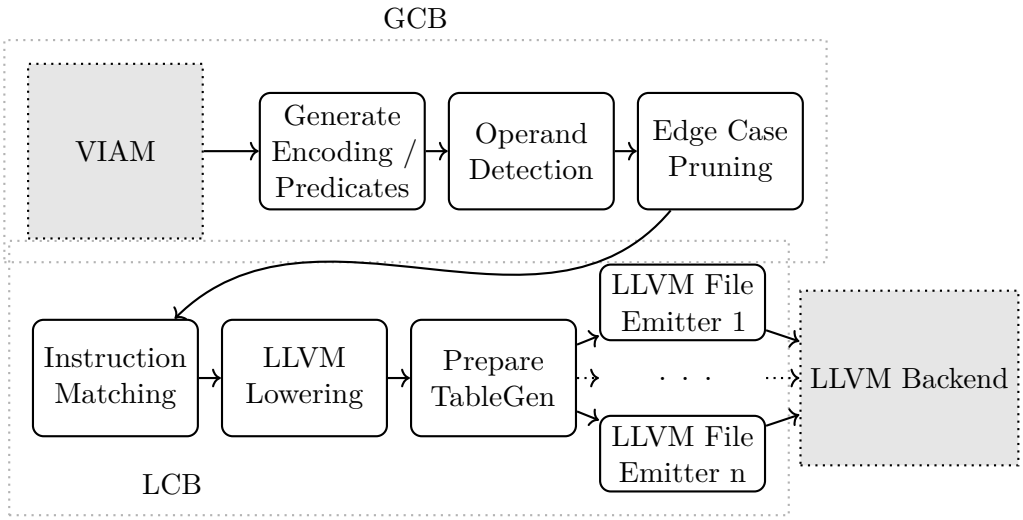

Figure 3.1: Compiler Generator Overview

VADL specification.

3.3 Encoding Generation

During the compilation, a compiler transforms a program into assembly or machine code. In VADL, each instruction has an *assembly* definition, shown in Listing 2.8, that is its human-readable representation. In contrast, the instruction's format defines the binary representation of the instruction, shown in Listing 2.7. As explained in Section 2.3.1, a field access function serves as a separation between instruction behavior and its decoding. However, a compiler encodes an immediate value into a format's field to emit machine code. The *encoding function* is the inverse function of the field access function.
To improve the readability of a specification, the compiler generator can automatically generate encoding functions in the following cases:

1. It is trivial to generate the encoding when the field access zero or sign extends a field. In that case, the encoding function truncates its operand to the field's bit width.
2. If the field access function shifts the value to the left, then the inverse function shifts the value to the right.
3. If the field access function uses addition or subtraction, then the inverse function is to invert the operations. The equations below show an example where $f(x)$ is the field access function's value and x the encoded field value. The inverse function is the computation of x based on the field access function. The variable C is an arbitrary constant.

$$\begin{aligned} f(x) &= x + C \\ \text{Let } y &= f(x) \\ y &= x + C \\ y - C &= x \end{aligned} \tag{3.1}$$

In the equation below the encoded value is negated and the left-hand-side has to be inverted.

$$\begin{aligned} f(x) &= -x - C \\ \text{Let } y &= f(x) \\ y &= -x - C \\ y + C &= -x \\ -y - C &= x \end{aligned} \tag{3.2}$$

When the compiler generator cannot generate encoding functions for more complex field access functions, encoding functions must be specified explicitly. Listing 2.9 shows an example where automatically generated encoding functions are manually added to the specification for the reader.

3.4 Predicate Generation

The field access function *immS* sign extends the field value, shown in Listing 2.7. Predicate functions are checked by the instruction selectors to verify that an immediate value's binary representation fits into the format's field. The predicate function is a boolean function that returns *true* when the immediate value fits, false otherwise. A machine instruction is selectable when all the used predicate functions are true. The term *used* refers to the fact that a format can have field access functions that are not used by any instruction.
As for the encoding generation, the compiler generator generates predicate functions automatically in the following cases:

1. It is trivial to generate the predicate when the field access zero or sign extends a field. In that case, the predicate checks whether the value fits into the underlying field.

2. If the field access function shifts the value to the left, then the lower bits must be zero, and the compiler generator adds a check to ensure the value is within the allowed range.

3. If the field access function uses addition or subtraction, then the predicate checks whether the value fits into the underlying field.

A predicate function must be provided in the specification when the mentioned cases do not apply. Listing 2.9 shows an example, where automatically generated predicate functions are manually added to the specification for the reader.

3.5 Operand Detection

Generally, every instruction has operands, which act as an input for the instruction. For instance, the instruction *addi a0, a1, $100* has three operands. $a0$ is the destination register operand, $a1$ is a source register operand and $100 is an immediate operand. In VADL, instruction operands are not explicitly defined. Therefore, the compiler generator has to automatically extract them from the instruction behavior graph. A *register file operand* becomes a *register operand* after the register allocation. A *bare symbol operand* defines a symbol, like an address, that is unknown at compile-time.
The GCB iterates over all the nodes in the instruction behavior graph and distinguishes the following three cases:

1. If the node is a *ReadRegTensorNode* or *ReadArtificialResourceNode* that reads from a register file in the behavior graph, then the node is promoted to a source register file operand.
2. If the node is a *WriteRegTensorNode* or *WriteArtificialResourceNode* that writes a register file in the behavior graph, then the node is promoted to a destination register file operand.
3. If the node is a *FieldAccessRefNode* in the behavior graph, then the node is promoted to an immediate operand or bare symbol operand.

The third case distinguishes immediate and bare symbol operands because VADL specifications treat addresses like immediate values. However, LLVM requires handling them differently because they must be updated by the linker. Addresses cannot be determined at compile-time. Instead, the compiler emits placeholders that are replaced by the linker, called relocations.

Moreover, instruction behavior graphs with nodes that read or write registers with constant indices, such as *X(11)*, are ignored as operand candidates because only register files with an immediate value as an index can serve as operands in LLVM. Instead, concrete registers such as the stack pointer, PC, or *X(11)* are treated as side effects of the instruction. The term *concrete register* is used for registers that are not part of a register file or register files with constant indices. A side effect is a register access that is not an operand of the machine instruction.
In other words, nodes in an instruction behavior graph, accessing a concrete register, will not become an operand because only register files or immediate values are operands in

LLVM. Instead, concrete registers have to be specified in `Use` or `Def` lists in TableGen, shown in Listing 2.3. For that reason, the compiler generator removes concrete registers from the instruction behavior graphs.

3.6 Instruction Classification

The compiler generator generates the machine instruction semantics from the instruction behavior graph. *Instruction classification* is the process of categorizing machine instructions based on their behavior. It is the key mechanism to effectively close the semantic gap by generating different instruction selection patterns for each classification label. A classification label is assigned when certain properties are fulfilled.

Table 3.1 presents classification labels and their properties. The table has two columns. When the properties in the second column are fulfilled then the first column's label is assigned to the machine instruction.

Generally, optimizations on the instruction behavior graph can increase the chances of a match. However, some properties query the unmodified instruction behavior graph because reads and writes to register files with constant indices and concrete registers are removed. Concrete registers are not considered by the instruction selection. Edge case pruning, described in Section 3.6.1, eliminates branches to match LLVM's undefined behavior.

Classification Label	**Properties for Classification**
ADD	Builtin *vadl::ADD* and two register file operands
ADDI	Builtin *vadl::ADD* and one register file operand and immediate operand
XOR	Builtin *vadl::XOR* and two register file operands
XORI	Builtin *vadl::XOR* and one register file operand and immediate operand
SLTH	Builtin *vadl::SLTH* and two registers file operands
...	Continue for other arithmetic and logical operations
SUBSC	• A builtin *vadl::SUBSC* with two register file operands • Unmodified behavior has the status flags: *negative*, *zero*, *overflow*, *carry*
BEQ	• Has exactly one *IfNode* • Unmodified behavior has no status flags • Writes the PC

...	Continue for other conditional branches
BEQ_BY_STATUS_REG	• Has exactly one *IfNode* • The conditional checks that the *zero* status flag is zero. • Writes the PC
BNEQ_BY_STATUS_REG	• Has exactly one *IfNode* • The conditional checks that the *negative* status flag is equal to the *overflow* status flag. • Writes the PC
BLT_BY_STATUS_REG	• Has exactly one *IfNode* • The conditional checks that the *negative* status flag is not equal to the *overflow* status flag. • Writes the PC
BGT_BY_STATUS_REG	• Has exactly one *IfNode* • The conditional checks that the *negative* status flag is equal to the *overflow* status flag and the *zero* status flag is equal to zero. • Writes the PC
BLEQ_BY_STATUS_REG	• Has exactly one *IfNode* • The conditional checks that the *negative* status flag is not equal to the *overflow* status flag or the *zero* status flag is equal to one. • Writes the PC
WRITE_MEM	• Has exactly one *WriteMemNode* • Has no *ReadMemoryNode* • Has an immediate
READ_MEM	• Has exactly one *ReadMemNode* • Has no *WriteMemoryNode* • Has an immediate
INDIRECT_JUMP (absolute)	• Writes the PC based on the value in a register. • Writes a register

JUMP_AND_LINK (relative)	• Writes the PC based on value computed with PC • Writes a register
JUMP	• Writes the PC • Has no writes to a register, register file or memory

Table 3.1: Properties for Instruction Classification

Table 3.1 contains two entries for loading from and storing to memory. Both *WRITE_MEM* and *READ_MEM* require an immediate value as an operand, which is added to a register operand. Other ISAs support memory instructions that allow more complex address computations. However, our compiler generator does not support these at the moment.

3.6.1 Edge Case Pruning

Section 2.4.2 introduced the terms semantic gap, defined and undefined behavior. The LLVM documentation, in [Anoh], differentiates between *immediate undefined behavior* and *deferred undefined behavior*. Immediate undefined behavior occurs when an undefined result is produced, e.g., operations that cause the CPU to raise exceptions. While deferred undefined behavior occurs when an undefined result is used. LLVM assigns a special value, called *poison*, to an undefined result.

To improve the instruction selection pattern generation and instruction classification, we introduce a technique called *Edge Case Pruning*. Generally, undefined behavior occurs in edge cases. The generator identifies edge cases and removes them from the behavior graph. However, verifying whether the pruned branch is undefined behavior creates a chicken-egg-problem. The instruction classification determines which branches have undefined behavior. For instance, an edge case, where the divisor is zero, is only undefined behavior for the division instruction. However, without edge case pruning, the instruction could not be classified in the first place. Consequently, the LCB can only verify whether the pruning method removed a branch correctly retrospectively. It is future work to enhance the edge case pruning.

The GCB distinguishes two cases:

1. If a branch of a conditional is raising an exception, it is an edge case.

2. If the condition of a conditional is checking values for concrete values, it is an edge case.

```
instruction DIV : Rtype =
    X(rd) :=
      if X(rs2) = 0 then
        0 as Regs // Edge case
      else
        X(rs1) / X(rs2)

instruction DIV2 : Rtype =
    X(rd) :=
      if X(rs2) != 0 then
        X(rs1) / X(rs2)
      else
        0 as Regs // Edge case

instruction DIV3 : Rtype =
    X(rd) :=
      if X(rs2) = 0 then
        raise {
            PC := ... // Exception Handler
        }
      else
        X(rs1) / X(rs2)
```

Listing 3.1: Division instructions for edge case pruning in VADL

Figure 3.1 shows multiple representations of a division instruction. The `DIV` instruction defines that the true branch has a narrow scope. While the `DIV2` defines the else branch to be the edge case. The third `DIV3` instruction raises an exception when the divisor is zero.

3.7 Lowering Machine Instructions

The term *Lowering Machine Instructions* refers to translating instructions, defined in VADL, to instruction definitions and selection patterns in TableGen. Listing 2.3 presents an instruction definition for RISC-V's *ADDI*.
Building on the classification presented in Section 3.6, machine instructions are split into three categories.

First, machine instructions, classified as arithmetic, logical, and comparative are considered to have no semantic gap. For these machine instructions, the VADL operations (*Builtins*) and LLVM IR instructions are semantically equivalent.

Second, machine instructions such as conditional and unconditional jumps, memory

stores, and loads are considered to have a semantic gap. For these, the compiler generator emits predefined selection patterns.

Finally, the last category contains machine instructions that are not classified. For instance, a fused multiply-and-add instruction has no instruction classification label. Machine instructions, from the first category or which have no classification label, generate selection patterns with the recursive generation method that is presented in Section 3.7.1. Whereas, machine instructions in the second category are emitted as predefined template TableGen patterns.

Figure 3.2 gives a high-level overview of combining instruction classification and machine instruction lowering. Each machine instruction goes through this process, checking the instruction behavior graph for the properties, presented in Table 3.1. The node *Check properties of dataflow* has three outgoing edges that correspond to the categories mentioned above.

Unfortunately, LLVM's instruction selectors have limitations, and generating invalid TableGen patterns cause the compiler to fail to compile. Figure 3.2 checks the instruction behavior graph for *red flags*, which stops the pattern generation for the machine instruction. A more detailed explanation is presented in Section 3.7.2.

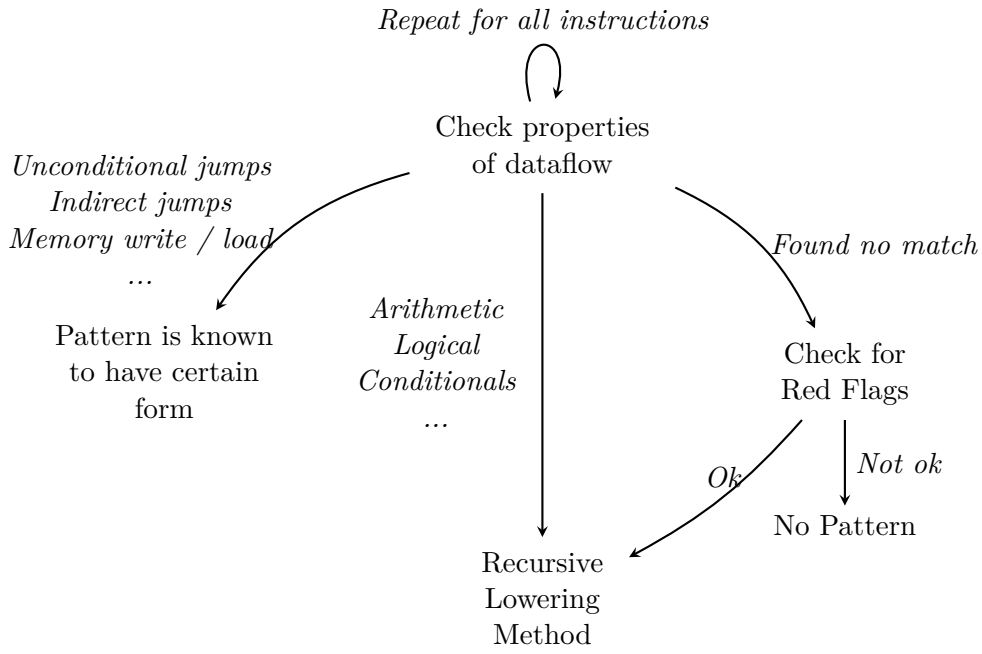

Figure 3.2: General Lowering Approach for Machine Instructions

```
def : Pat<(add X:$rs1, X:$rs2),
    (ADD X:$rs1, X:$rs2)>;
```

Listing 3.2: TableGen Selection Pattern for the RISC-V *ADD* instruction

3.7.1 Recursive Instruction Pattern Generation Method

Figure 2.4 shows the instruction behavior graph of RISC-V's *ADD* instruction. The instruction behavior graph is a multi-graph, containing both CFG and DFG. LLVM's instruction selectors are limited to tree patterns and cannot handle complex control-flow. The mechanism presented in Section 3.7.2 introduces checks to verify that a graph is lowerable before applying this method. The underlying assumption is that VADL's operations and SelectionDAG's nodes correspond one-to-one. The algorithm, shown in Figure 3.2, traverses recursively through the dataflow, starting with the *Side-Effect* nodes, and maps the VADL operation nodes to an equivalent SDNode. The generated instruction selection pattern is shown in Listing 3.2.

3.7.2 Red Flag Mechanism

As stated in Section 3.7, generating invalid selection patterns might cause LLVM compilation errors. The red flag mechanism is a filter that checks the instruction behavior graph before the recursive instruction pattern generation method, presented in Section 3.7.1, is applied. A *red flagged* machine instruction skips the selection pattern generation. Checks vary for each classification label but the following conditions are supposed to give the reader an overview.

1. Does the instruction behavior graph have an unsupported builtin?
2. Does the instruction behavior graph have multiple outputs?
3. Does the instruction behavior graph read or write from memory? This check is omitted for some classifications labels like simple loads and stores.
4. Does the instruction behavior graph have a select node?
5. Does the instruction behavior graph have sign or zero extension nodes?
6. Are all the instruction's operands used?

3.7.3 Considerations

The recursive instruction pattern generation method, presented in Section 3.7.1, is similar to the idea presented in [HL10, p72]. In a lot of cases, nodes can be replaced one-to-one when VADL operation semantics match SDNode semantics.
For instance, *truncstore* is an SDNode where a truncate and store are merged. The LCB

must merge nodes to establish a one-to-one mapping because VADL has no combined operation.

SDISel differentiates between an immediate value and a basic block. Immediate values are constants with a fixed type. A basic block is an address whose value is unknown at compile-time. SDISel has different types for both. However, VADL has no concept of basic blocks. Consequently, the LCB needs to automatically differentiate between a machine instruction that uses an immediate value or a basic block because both are represented with a field access function in the VADL specification. The LCB replaces nodes with field access functions, with specialized basic block nodes for machine instructions, identified as branch instructions in the instruction classification.
Similarly, immediate values are replaced by symbol nodes for instructions that are identified as jump instructions in the instruction classification.

To summarize, before the lowering, the LCB merges and replaces behavior graph nodes to bridge the gap between LLVM IR and VADL.

3.8 Lowering Pseudo Instructions

Pseudo instructions are aliases for one or multiple machine instructions that increase the readability in the assembly representation. Listing 3.3 shows a pseudo instruction that loads an immediate value into a register in RISC-V. It is expanded into a sequence of two machine instructions that load the most significant bits and, afterwards, add the least significant bits to the destination register *rd*. Each machine instruction calculates the immediate value with a relocation function. Operands that are constants can be calculated at compile-time, but addresses are replaced with relocations.

3.8.1 Operand Detection for Pseudo Instructions

Operation detection for pseudo instructions is more difficult than for machine instructions. Listing 3.3, in line 1, shows a pseudo instruction that has two operands. Operand detection for pseudo instructions is to determine how an operand is *used*. An operand can be used as a register index or a field access function. Additionally, an operand must be classified as source or destination operand. For that particular pseudo instruction, the operand *rd* is the destination register because both `LUI` and `ADDI` assign the operand *rd* to the machine instruction's field *rd*, which is an output operand for each of them. However, the machine instruction `ADDI` also assigns the destination operand *rd* to the field *rs*1, which is an input operand.
When an operand is both an input and an output operand, a conflict arises. The LCB resolves the conflict by ignoring the input operand. Therefore, the pseudo instruction `LI` has the *rd* operand as the destination operand.

Another issue is the *symbol* operand in Listing 3.3 in line 1. It is unclear whether the

```
1 pseudo instruction LI(rd : Index, symbol : Bits<32>) = {
2     LUI  { rd = rd, imm = hi(symbol) }
3     ADDI { rd = rd, rs1 = rd, imm = lo(symbol) }
4 }
```

Listing 3.3: Loading an immediate for RISC-V in VADL

symbol operand is a value known at compile-time or link-time. The pseudo instruction expansion emits C++ code for both cases.

Specifications, written in VADL, specify machine and pseudo instructions in binary representation, which is not suitable for the compiler generation. In Listing 3.3, the pseudo instruction assigns its operands to the machine instructions' fields. However, each field has an encoding, and the pseudo instruction's operands must be encoded before the assignment during the expansion. A field's encoding may depend on multiple field access functions. Internally, the LCB has to create an intermediate operand list that is encoded afterwards.

3.9 Lowering Registers

LLVM IR is in SSA and has an unlimited number of registers, while a target-machine has only a limited number of physical registers. In LLVM, registers are grouped in a register class, similarly to a register file. Target architectures typically have a register hierarchy in which each register is composed of smaller subregisters. An index defines the access relationship between register and subregister [Col25, p330-331].

The process of defining the physical registers in the compiler is called *register lowering*. In particular, the **`register`** definitions in VADL specifications are transformed into LLVM register definitions. Listing 3.4 presents a register definition of a concrete register and register class for the AArch64 ISA. Other subregister definitions and register classes are omitted for brevity. As mentioned above, the register class is a compiler abstraction for register files. In LLVM, each register is modeled explicitly and grouped in a register class that, additionally, defines the allocation sequence. The allocation sequence is the preferred order for the register allocation. The LCB uses the caller-saved, callee-saved, and then the remaining registers for the allocation sequence. If there is no calling convention for the register class in the ABI then the LCB orders the registers in ascending order based on the index in the register file.

Line 1-3 in Listing 3.4 defines multiple `SubRegIndex` which are the indices for the subregisters. The subregister relationship is defined in line 8 with the index given in line 9. Line 7 defines the name that is printed for the assembly printing. Line 10

```
1  def SUB_32   : SubRegIndex<32>;
2  def SUB_32_1 : SubRegIndex<32>;
3  def FULL_64  : SubRegIndex<64>;
4
5  def S0 : Register<"S0">
6  {
7      let AsmName = "X0";
8      let SubRegs = [ R0, X0, W0 ];
9      let SubRegIndices = [ SUB_32, FULL_64, SUB_32_1 ];
10     let HWEncoding{4-0} = 0;
11     // ...
12 }
13
14 defvar aarch64temp = DefaultMode;
15
16 def SLenRI : RegInfoByHwMode<
17       [ aarch64temp ],
18       [RegInfo<64,64,64>]>;
19 def S : RegisterClass
20 < /* namespace = */ "aarch64temp"
21 , /* regTypes  = */  [  i64 ]
22 , /* alignment = */ 64
23 , /* regList   = */
24   ( add S9, S10, S11, S12, S13, S14, S15, S19, S20, S21, S22, S23,
25   S24, S25, S26, S27, S28, S0, S1, S2, S3, S4, S5, S6, S7, S8,
26   S16, S17, S18, S29, S30, S31 )
27 > {
28   let RegInfos = SLenRI;
29 }
```

Listing 3.4: Register definition for AArch64 in TableGen

defines the hardware encoding which is the value for the format's field in the binary representation. Line 19 defines the register class `S`. Line 24 defines the allocation sequence for the register class. Instruction selector patterns use the register class as type and the register allocator will assign a free register during compilation.

3.10 Relocations

The compiler generator differentiates between two kinds of relocations: user-defined and automatic. User-defined are defined with the keyword **`relocation`** in the VADL specification, shown in Listing 2.11.
Automatic relocations are generated by the LCB. These are generated for all machine instructions that have an immediate operand. PC accesses might only be resolved at

link-time and require patching by the linker.

Unfortunately, relocations prevent the generated compiler from emitting object files that are linkable by an *upstream* linker. The standard, in [Ano25b], defines which instruction formats the linker can update. However, the LCB generates more relocations than the standard defines [Ano25b]. The LCB inspects each instruction individually and generates a relocation if necessary. Even if the LCB generates only relocations for instruction formats, the generated compiler would still not be able to produce object files linkable by an upstream linker. RISC-V supports relocations for store instructions, which the LCB cannot generate because VADL cannot specify an instruction for `sw x11, %lo(global)(x5)`. The term *upstream* refers to other publicly available RISC-V linkers.

3.11 Instruction Expansion

A compiler frontend might generate an LLVM IR with an operation that the processor does not support. To address this mismatch, LLVM provides an expansion mechanism. A compiler backend can define which nodes should be automatically expanded into other operations before legalization [Anon].

The compiler generator has a list of SDNodes where each node must be the root of at least one instruction selection pattern after the patterns are generated. If no pattern covers that particular node, then it is likely that a selection error occurs at run-time because the compiler cannot transform an IR operation into the corresponding machine instructions.

For uncovered SDNodes, the LCB can force LLVM to automatically expand the nodes into different nodes. However, this approach is incomplete, since not all nodes are expandable. In some cases, it can even lead to infinite loops when two nodes expand into each other. But overall, automatic expansion reduces the possibility of selection errors. [Hsu24] discusses the shortcomings of LLVM's legalization in greater detail.

3.12 Constant Materialization and Register Adjustment

During compilation, constants in LLVM IR are transformed into machine instructions. This transformation is called *constant materialization*. Machine instructions have a limited bit width, and the size of constants might vary. For that reason, it is possible to define multiple `constant sequence` in VADL. For instance, a constant with a value between -2048 and 2047 can be materialized with the *ADDI* instruction. Larger constants use the combination of *LUI* and *ADDI* for materialization. Listing 2.12 defines an ABI with three constant sequences, differentiated by their second argument's type. The LCB sorts the sequences by the type and the compiler will prefer sequences with a smaller type first.

```
[ only negative ]
constant sequence(rd: Bits<5>, val: SInt<32>) =
{
  MOVZPos00 { rd = rd, imm16 = (val & 0xFFFF) as Bits<16> }
  MOVKWPos16 {
    rd = rd,
    imm16 = ((val >> 16) & 0xFFFF) as Bits<16>
  }
  SUBX { rd = rd, rn = 31, rm = rd }
}
```

Listing 3.5: Constant Sequence for AArch64 in VADL

Register adjustment sequences are similar to constant sequences. However, constant sequences overwrite any existing value in the destination register. Constant sequences require an additional register to materialize the constant first to avoid overwriting a destination. Register adjustment sequences are an optimization in which an existing register is provided as a source register operand.

For AArch64, an extension for constant sequences is required. Specifically, VADL has an annotation **only negative** for **constant sequence**, shown in Listing 3.5. AArch64's machine instructions *MOVZ* and *MOVK* only allow unsigned values as immediate operands. Consequently, the compiler generator has to add an additional check and select the appropriate constant sequence. The compiler uses the *val* variable with an unsigned value, however, the register value is inverted in the last instruction because the sequence is only used for negative values.

3.13 Conditional Branch and Compare-and-Jump Instructions

RISC-V has machine instructions for comparing two registers and perform a jump based on the result. Both comparison and jump are performed together by one machine instruction. In contrast, AArch64 has realized this in two separate machine instructions. One instruction for the comparison, and one instruction for the jump. The compiler generator's challenge is to guarantee that the compiler always schedules both instructions together.

Based on classification labels, presented in Section 3.6, the compiler *chains* together instructions when no classification label *BEQ* exists. The compiler generator overwrites the SDNode lowering of the *BR_CC* node to emit both AArch64's machine instructions to perform the conditional jump [Ano25a].

CHAPTER 4

Problems

This section elaborates problems and shortcomings of the current implementation that was presented in Chapter 3.

4.1 Recursive Lowering

A method for generating an instruction selector pattern was presented in Section 3.7.1. A pattern is generated by recursively traversing the dataflow. The recursive lowering method requires a one-to-one mapping between VADL's operations and SDNodes to bridge the semantic gap, presented in Section 2.4.2.

In the ISA of AArch64, the addition machine instruction's register operand can be shifted by an immediate value. By using the recursive generation method, a tree pattern is generated that selects a shifting operation in the LLVM IR. Listing 4.1 shows two instruction selection patterns as an example. The first instruction selector in line 2 is an addition with two registers. This mirrors how a compiler frontend emits LLVM IR for an addition of two values. However, the instruction selector cannot automatically create a zero immediate value for the second pattern in line 5.

To avoid selection errors during compilation, in some cases VADL specifications must define instructions so that the LCB generates selectable instruction selection patterns. The mapped instructions *ADDX* and *ADDXLSL* are both the same AArch64 machine instruction in Listing 4.1. However, the instruction *ADDX* has no shifting operation in the behavior, defined in the VADL specification.

```
# Simple addition
def : Pat<(add X:$rn, X:$rm),
   (ADDX X:$rn, X:$rm)>;
# Addition with shift
def : Pat<
   (add X:$rn, (shl X:$rm, AArch64Base_ADDXLSL_imm6AsInt8:$imm6)),
   (ADDXLSL X:$rn, X:$rm, AArch64Base_ADDXLSL_imm6AsInt8:$imm6)>;
```

Listing 4.1: Addition instruction selection patterns in AArch64

```
def : Pat<(add X:$rs1, ADDI_immSAsInt8:$immS),
   (ADDI X:$rs1, ADDI_immSAsInt8:$immS)>;
```

Listing 4.2: ADDI instruction selection pattern

4.2 Frontend and Backend Type Mismatch

A compiler frontend generates LLVM IR, transforms it into a SelectionDAG, and the backend selects machine instructions. The compiler frontend generates LLVM IR based on a source language like C. On the other hand, the LCB generates instruction selector patterns based on the VADL specification. The types for the patterns are generated from the instruction's behavior types.

Listing 4.2 shows an example of an instruction selector pattern for an addition with an immediate value that has eight bits. However, a compiler frontend might generate LLVM IR for different types, e.g., 32 bits. In that case, the instruction selector cannot match the pattern and loads the immediate value into a register that is then added to the register.

To summarize, the problem is that the types for the instruction selection patterns are determined by the types in the instruction behavior graph. However, the compiler frontend might generate larger types, resulting in inefficient instruction selection.

4.3 Truncation

Truncations are used to cut off the most significant bits, reducing the type's size. Both VADL and LLVM have truncations. In Listing 4.3, the second operand $X(rs2)$ reads from the register file in the *ADDI5* instruction. Reading a value from $X(rs2)$ returns 64 bits, but a VADL specification truncates the value to 5 unsigned bits. The recursive

```
1     instruction CUSTOM : CUSTOM_FORMAT =
2         if(X(rs1) == X(rs2)) then
3             X(rd) = X(rs1)
4         else
5             X(rd) = Y(rs1)
6         end
```

Listing 4.4: Fictive problematic instruction because register index is used with multiple register files

instruction selection pattern generation method, presented in Section 3.7.1, emits a truncation node in the selector pattern. However, a compiler frontend does not emit a truncation instruction for the addition in LLVM IR.

To generate instruction selection patterns, the LCB ignores truncation nodes and will not emit them.

```
1 instruction ADDI5 : Rtype =
2     X(rd) := (X(rs1) as Bits<64> + X(rs2) as Bits<5>)
```

Listing 4.3: ADD instruction with a 5 bit operand in VADL

4.4 Register index with different register files

Listing 4.4 presents a machine instruction that references two different register files. The problem is that the type is ambiguous for input operand $rs1$. The LCB could create two different input operands for both register classes, however, it is not clear how these can be merged into the instruction encoding. Therefore, the LCB cannot accept a VADL specification with such a behavior.

4.5 Unused operands for pseudo instructions

Section 3.8 discusses the operand detection for pseudo instructions. The problem is that the GCB analyzes the machine instructions' operands to detect the type of the operand for the pseudo instructions. However, an operand, which is not used by a machine instruction in the pseudo instruction, but is nonetheless referenced in the assembly printing, has no type, and an instruction definition cannot be generated.

Listing 4.5 shows a pseudo instruction with a machine instruction that does not

```
pseudo instruction XXX(rd : Index, symbol : Bits<32>) = {
    FOO  { rd = rd, rs1 = rd }
}
assembly XXX = (mnemonic, " ", hex( symbol ))
```

Listing 4.5: Pseudo Instruction with an unused operand

use the *symbol* operand. However, the operand is used in the assembly printing which is problematic because it is unclear what the type of the operand is.

4.6 Instruction Alignment

Instruction alignment refers to the positions of instructions in memory. This becomes especially important when formats do not have the same size or have a variable length. Unfortunately, the LCB does not align instructions, which can reduce performance or cause exceptions during execution.

4.7 Type Mismatches during Instruction Expansion

Section 3.11 explained that LLVM can automatically map SDNodes into other SDNodes. This is necessary because processors might not support all operations provided by programming languages. During the development of the LCB, we faced the issue that the *DAGCombiner* creates new immediate values with a larger type that our machine instructions supported. For instance, *DAGCombiner* created a logical-right-shift SDNode where the destination and source register file operands are 32 bits integers but the immediate operand has a 64 bit integer type. A specification in VADL has to provide an instruction to cover such a pattern, otherwise, instruction selection fails.

4.8 Free Truncations and Extensions

In AArch64, 64 bit registers have 32 bit subregisters. Therefore, truncation nodes from 64 to 32 bits do not require explicit instructions, as subregisters already provide that for free. Similarly, a zero extension from 32 to 64 bits is free when using the full register. However, truncation is unsafe when multiple equivalent subregisters are used, as in the example 3.4. In the implementation, the truncation SDNode is replaced with a subregister index. However, multiple register files introduce ambiguity, and the compiler might crash if the subregister index is applied to the wrong register file as implemented in [Anoc].

4.9 Application Binary Interface

The LCB generates a calling convention in TableGen, given a VADL specification with an ABI section. While TableGen is quite convenient for a generic calling convention, more complicated calling conventions cannot be expressed effortlessly. For these cases, upstream compiler backends use C++ to handle different edge cases. Moreover, many edge cases are not handled by the compiler backend at all, but by the compiler frontend. The LCB's support for calling conventions is limited only to TableGen. Additionally, Nikita Popov identified a few shortcomings in [Pop25], a proposal for better ABI support in upstream compilers.

CHAPTER 5

Evaluation

This chapter presents the evaluation of three compilers that were generated by the LCB. Each of them is compared with the upstream compiler which is handwritten and considered to be the industry standard.

5.1 RISC-V

This section presents the evaluation for RV32IM and RV64IM. The LCB's compiler has no libc for the executables, and the object files do not have the same relocations, which makes the generated binary files incompatible. For performance evaluation, the compiler prints the assembly and then uses GCC's assembler and linker to produce an executable. The presented evaluation uses position-independent code. The Embench suite was used for benchmarking [Anob]. All the benchmarks were compiled with $-O3$. As performance metric we used the number of executed instructions during a benchmark run. Figure 5.1 demonstrates that a program compiled by our compiler for RV32IM executed 29.73% more instructions on average than upstream, while Figure 5.2 shows that, for RV64IM, 43.83% more instructions on average were executed. All benchmarks were executed with an upstream QEMU machine emulator [Bel05]. The Embench suite has 22 benchmarks. For RV32IM, we executed successfully 21 benchmarks. The benchmark *cubic* triggers a miscompilation. For RV64IM, we executed successfully 20 benchmarks. The benchmark *wikisort* triggers a miscompilation and the benchmark *slre* causes a compiler crash because two operands of a phi node have different types, violating an internal invariant.

The following cases were identified as the main reasons for the performance difference:

1. We identified that constants are not moved out of loops, i.e, the loop invariant code motion seems to fail. This adds a significant overhead in the evaluation represented in Figures 5.1 and 5.2. In general, most programs' execution is spent in loops.

2. Section 4.2 discusses the problem of the mismatching immediate types between compiler frontend and backend. Our performance evaluation showed that we encountered this issue for *SLLI*. The compiler failed to use the instruction *SLLI* because the IR's constant had a larger type. Therefore, the compiler had to load the constant into a register and had to fallback to *SLL*. It is our opinion that preventing this fallback would have significant improvements and reduce register pressure.

3. We noticed that the LCB's compiler generates assembly code with more spills than upstream. We think that the previously mentioned problem with *SLLI* increases register pressure.

4. RISC-V supports constant offsets for addresses in store instructions. However, the LCB's compiler does not support an offset addressing mode. Therefore, additional instructions are generated to compute the effective address.

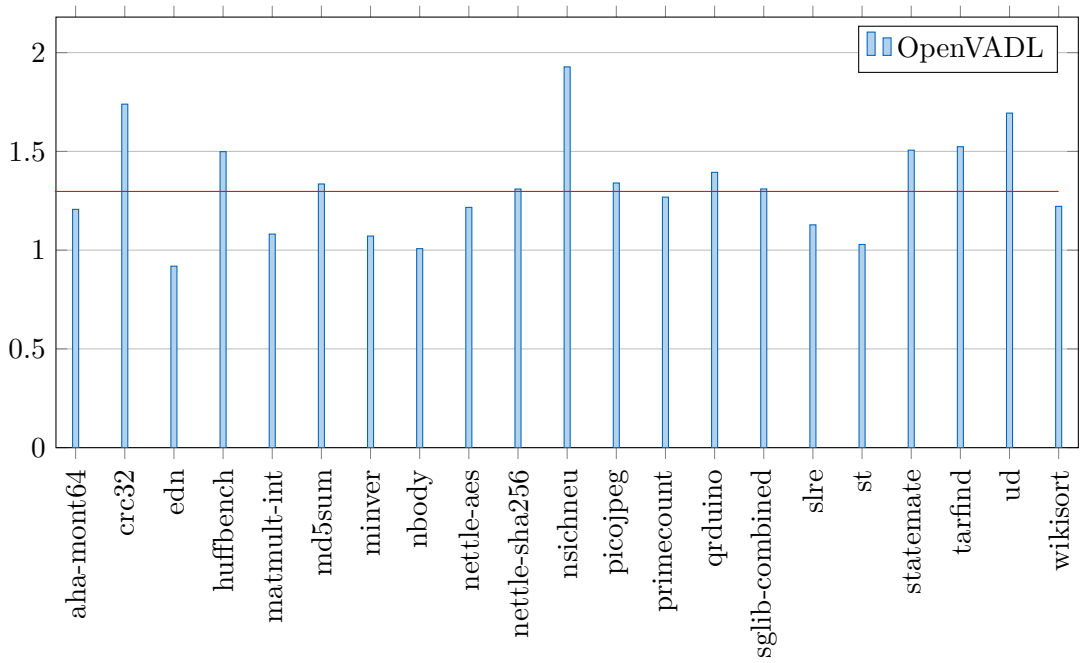

Figure 5.1: OpenVADL LCB RV32IM: Embench QEMU / relative number of executed instructions compared to upstream (lower better)

For comparison, the previous implementation of the compiler generator was limited to generate a compiler for RV32IM and was unable to compile with the optimization level $-O3$ [Fre+].

A compiler is generated in 0.21 and 0.20 seconds for RV32IM and RV64IM ($n = 5$, MacBook Air M2 - Sequoia 15.6.1, Graal CE 25.0.1, OpenVADL Commit 8a86982 on

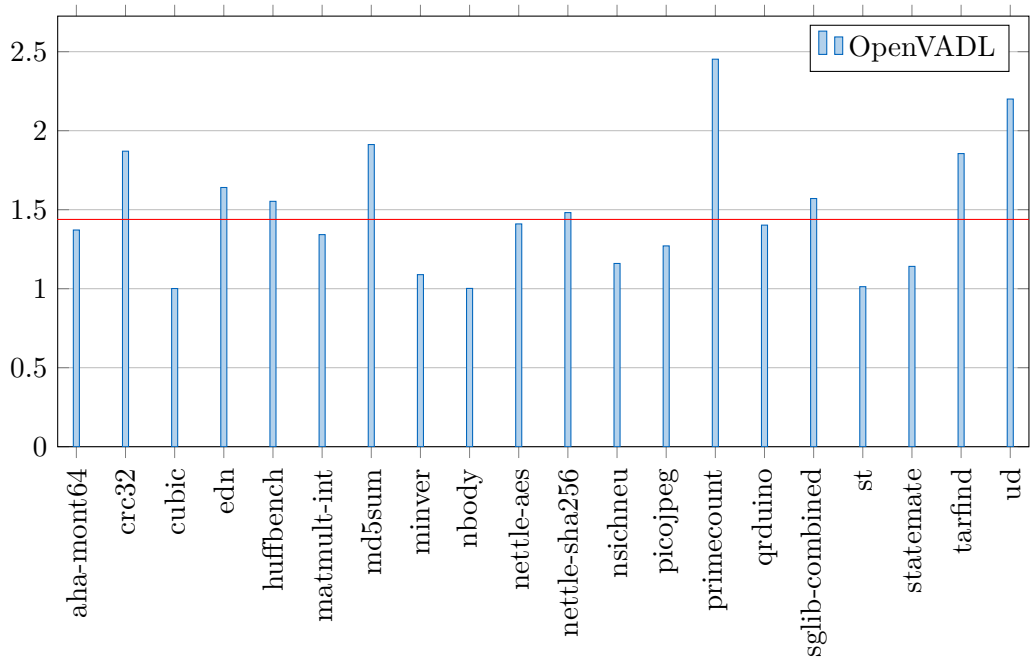

Figure 5.2: OpenVADL LCB RV64IM: Embench QEMU / relative number of executed instructions compared to upstream (lower better)

24.11.2025), respectively. The compiler generator was built with Graal VM's native compile feature. The generated files have a total size of 2.6MB and 2.7MB for RV32IM and RV64IM, respectively.

5.2 AArch64

In contrast to the RISC-V compiler generated by the LCB, the AArch64 compiler is not fully functional yet. Therefore, we cannot present a complete performance evaluation. We can compile and execute programs that use arithmetic, comparison, and logical instructions. However, global variables, jump tables, and indirect jumps are not supported yet. Figures 5.1, 5.2, and 5.3 represent the code snippets that the compiler fails to compile. The resulting expression is negated to return a successful exit code for verification.

First, the compiler crashes with a stack trace pointing to function *llvm::InstrEmitter::ConstrainForSubReg*. This indicates that the compiler has a bug in its handling of sub-registers.

Finally, the generated AArch64 code lacks a selection pattern for logical-shift-right with a 64 bit immediate operand. This causes a compilation error. The problem is that AArch64 has no machine instruction for computing the remainder of two registers.

```c
typedef int (*func_t)( int );

static int driver( int a);

int simple(int a) {
    return a;
}

int main( void )
{
    return !(driver( 0 ) == 0 && driver(2) == 2); // => 0
}

static int driver( int a )
{
    func_t f_sub = &simple;
    return f_sub(a);
}
```

Listing 5.1: Indirect jump example

As a consequence, the remainder operation is expanded to different instructions by LLVM. This expansion is not covered by a pattern and causes a crash.

A compiler is generated in 2.77 seconds on average ($n = 5$, MacBook Air M2 - Sequoia 15.6.1, Graal CE 25.0.1, OpenVADL Commit 8a86982 on 24.11.2025) and the total size of the generated files is 25MB. The compiler generator was built with Graal VM's native compile feature.

```c
int option1() {
  return 0;
}
int option2() {
  return 1;
}
int option3() {
  return 2;
}

// Function pointer type
typedef int (*JumpFunction)();

int main() {
    // Array of function pointers (jump table)
    JumpFunction jumpTable[] = {
        option1,   // Index 0
        option2,   // Index 1
        option3    // Index 2
    };

    return !(jumpTable[0]() == 0 && jumpTable[1]() == 1
        && jumpTable[2]() == 2);
}
```

Listing 5.2: Jump table example

```c
int ArrayA[] = {
            1
   };

int main() {
    return !(ArrayA[0] == 1);
}
```

Listing 5.3: Global variable example

CHAPTER 6

Related Work

6.1 LISA

LISA is a PDL to formalize processor architectures. It has two main elements: resources and operations. Resources define memory, registers, and pipelines. Operations define the instructions that are supported by the processor [Pee+99; HML02]. Like VADL's instructions, LISA's operations also define assembly printing and behavior. It is a mixed language, therefore, it contains both behavioral and structural information about the processor. While VADL generates an LLVM compiler, LISA uses the compiler *CoSy*. Like the LCB, LISA's compiler is also a tree matching compiler. Therefore, it cannot deal with multiple output instructions. But according to Sandro Rigo, LISA's main contributions are the description of pipelines and the sequencing model [Rig+04, p2]. In [Cen+04; Cen+05a] [HL10, p58, p59], an extension added the keyword *semantics* to LISA. In comparison, LCB's goal is to derive the semantics from the instruction behavior automatically. In [Cen+05a], the authors acknowledged that the C language does not change and, therefore, the IR elements required for instruction selection are fixed. We think that this idea is very similar to the instruction classification, presented in Section 3.6. However, the LCB does not differentiate between mapping types as described next. LISA's approach for lowering the semantics to machine instructions is described in [Cen+05a] [HL10, p72]. LISA differentiates between multiple mapping types.

- **One-to-One Mapping.** The IR and the target machine instruction's semantics are equal. In that case, a mapping can be created effortlessly. In general, arithmetic and logical instructions can be mapped directly because target ISAs usually support addition, subtraction, and shift instructions.
- **One-to-Many Mapping.** Such a mapping exists when a single IR element is transformed into a sequence of target instructions. An example of a *semantic transformation* is given in [Cen+05b] [HL10, p78] where a *neg* instruction is transformed into a sequence of *not* and *add*. The LCB has no concept of unmapped

rules. Therefore, OpenVADL's compiler generator has no knowledge when it has to apply such a transformation. At the moment, the LCB derives alternative mappings based on existing patterns. In other words, the LCB mutates existing patterns to cover a missing case when a target instruction is not matched. For example, the RISC-V ISA has no machine instructions for all possible conditional branch instructions. However, it is possible to derive the missing instructions from the existing conditional branch instructions. We refer to them as *alternative patterns*. Alternative patterns are always derived from the base pattern, because it is the instruction selector's task to choose the instruction with minimal cost. The previously mentioned semantic transformation for *neg* would not be covered because the LCB is missing the concept of mandatory patterns. Alternatively, it would be possible to extend the legalization to expand this IR element.

- **Many-to-One Mapping.** These mappings are important for ASIPs because they are used to achieve acceleration by fusing multiple instructions together. One example of such an instruction is the *fused-multiply-add* instruction. Such an operation is common for digital signal processors. The LCB uses the recursive lowering, presented in Section 3.7, to generate arbitrary tree patterns for fused instructions. According to [HL10, p79], the generator knows the mapping for each IR element individually and can create a corresponding tree pattern for LISA.

While we know that a poster was presented on an LLVM compiler backend generated by LISA at the LLVM Conference 2013 in San Francisco [Dob13], we are not aware of any other publication providing more details.

6.2 ArchC

ArchC is a PDL based on SystemC [Rig+04]. After its inception, a compiler generator was added in [ACB12] called *ACCGen*. ACCGen is also a LLVM-based compiler. However, LLVM's TableGen as code generator was not used. Furthermore, compiler backends often share many similarities, and the authors created a target-independent compiler backend. Auler, Centoducatte, and Borin acknowledge the same challenge by stating: "However, automatically inferring instruction semantics out of the aforementioned instruction behavior can be a very hard task." [ACB12]. The major difference between [ACB12] and the LCB is that ACCGen has an additional language element to specify the instruction selection pattern. The compiler generator's goal is to generate selection patterns from the instruction behavior graph. By specifying the instruction semantics, the authors' generator does not suffer from the same shortcomings presented in Chapter 4.

6.3 xADL

xADL is a language that was developed by Florian Brandner, Dietmar Ebner, and Andreas Krall at the TU Vienna [Bra09] [BEK07] [BPK13]. It is a structural language where the instruction's behavior and the MIA are mixed. Therefore, the processor designer can

specify exactly what happens at each pipeline stage. One of the generated compilers is based on LLVM. Since xADL is a structural language, instructions are not explicitly defined. Consequently, the ISA needs to be extracted. In [Bra09, p68], Brandner explains that the extraction is done by combining the micro operations of the dataflow through the netlist. The terms machine instruction and micro operation are used interchangeably in this work.
The generator then builds a tree grammar from the behavior. The tree grammar contains two types of rules. Each rule is a tree pattern that covers a tree fragment in the IR. Registers and immediates become nonterminals, while operations in the IR become terminals. Finally, the generator creates a mapping rule by converting the extracted micro-operations into a tree representation. The second rule type is a conversion rule that represents the conversion between registers at no cost.
In our view, the idea of the mapping rule generation corresponds to the recursive lowering approach, presented in Section 3.7. Also, the paper mentions templates and specializations. Those are necessary because some operations are not supported by xADL, such as sign extension [Bra09, p104]. However, the generator checks each template's requirements and replaces it if necessary. Regarding templates and specializations, we see similarities with the LCB's alternative patterns, where existing instruction selector patterns are rewritten based on the instruction classification, presented in Section 3.6.

6.4 Original VADL Compiler Generator

OpenVADL is the open source successor of a proprietary VADL implementation. The initial implementation of the compiler generator was developed by Alexander Graf in [Gra21] and it was later extended by Christoph Hochrainer in [Fre+]. However, the current implementation was developed from scratch.

The instruction classification, presented in Section 3.6, is built on top of the idea of Graf's instruction and pattern analyzer. Each instruction was categorized into branch, load, store and general instructions. This work extended the analyzer with a finer-grained classification algorithm and more functionality during the instruction lowering. In the initial implementation 48 additional instruction selector patterns and 300 lines of custom C++ were added manually for Graf's evaluation. Our assumption is that the customization was necessary to achieve the highest optimization level for the benchmark evaluation. Freitag et al. improved the original VADL compiler generator [Fre+]. However, it was limited to generating a compiler for RV32IM. Moreover, the generated compiler failed to compile the Embench suite with the *-O3* optimization level. In contrast, a compiler generated by the LCB does not need any manual customizations for RV32IM and RV64IM, demonstrating the improvements in the classification and generation in this work.

6.5 VEGA

VEGA is an AI-driven system to automatically generate a compiler backend [Zho+25]. The authors noticed that creating a backend involves customizing function templates to meet requirements. Its architecture is split into code-feature mapping, model creation, and target-specific code generation. First, it collects existing implementations for different architectures by scanning GitHub. Second, a large pre-trained code model, *UnixCoder*, is fine-tuned using the templates and extracted feature vectors. The model learns to transform target-independent code into target-specific code. Finally, VEGA generates a target-specific compiler backend given a new target's description. However, it is not clear what exactly this new target's description is. According to them: "Fig. 4(a) - Fig. 4(d) show how VEGA generates the RISC-V implementation of getRelocType. As depicted in Fig. 4(a), VEGA utilizes RISC-V target description files, such as RISCVInstrInfo.td and RISCVFixupKinds.h from TGTDIRs". It is not clear to us whether the generator is able to generate TableGen. In other words, we are not sure what the input for the generation is. The authors reported accuracies of 71.5%, 73.2%, and 62.2% for RISC-V, RI5CY, and XCORE, respectively. Therefore, the generated compiler still required manual effort to fix the generator's shortcomings. The fixes had to be done by experienced compiler engineers. In general, PDLs aim to avoid manual adaptations. The specifier expects that the generated compiler will work correctly when it is parsed and type-checked by the generator's frontend.

6.6 Completeness and Translation Validation

In [Bra10], Brander presented a method for developing a *complete* instruction selector. An instruction selector is complete when it can always generate machine instructions given any IR by a compiler frontend. It is even more challenging to achieve completeness when predicates are used. The traditional approach is to use finite tree automatons, but it is not possible to represent predicates in those. Brandner presented a technique to use dedicated *terminal symbols* to represent predicates. Additionally, he presented a rule generator to prove the completeness of an instruction selector.

While Brandner's approach ensures that an instruction selector is complete, it does not guarantee the correctness of the translation. Similarly, the LCB generates a compiler with heuristically generated instruction pattern selectors. In the worst case, the generator could swap addition and subtraction, resulting in a miscompilation for a program compiled by a generated compiler.

Addressing correctness, Lopes et al. presented a translation validation tool that checks whether a transformation is correct for LLVM IR. In [Ber+25], Berger et al. went one step further and verified that the generated machine code is correctly selected through lifting machine instructions into LLVM IR and checked whether the transformation is correct. Whereas Alive2 uses *Satisfiable Modulo Theories*, Lezuo presented a prover that

uses *direct symbol execution* to create a first-order representation of a program [Lez14]. Lezuo developed a language which is based on Abstract State Machines (ASM) to specify the semantics of the source language and machine instructions.

CHAPTER 7

Future Work

Chapter 4 outlined open problems aiming to improve the performance of a LCB generated compiler. It is our opinion that frontend and backend type mismatches discussed in Section 4.2 are unlikely to be solved automatically. Addressing these issues will require extensions to the VADL language.

Moreover, in Section 4.3, we presented the removal of truncations to bridge the semantic gap between compiler frontend and backend in the instruction selection. However, the implications for our AArch64 compiler remain uncertain. The AArch64 ISA has a lot of instructions that are generated into tree patterns for the instruction selection.

We presented the distinction between a GCB and LCB in [Fre+]. While this distinction still exists, the GCB is quite thin. Adding another compiler would require refactoring the LCB. In the author's opinion, the required effort would be considerable high, while the benefits of multiple compiler toolchains are low. In general, maintaining compatibility between different upstream compiler toolchains is a challenging task for compiler engineers. Consequently, resolving compatibility issues in automatically generated compilers is unlikely. The term *compatibility* means that multiple compilers can be used interchangeably. This requires that ABIs, relocations and libc implementations work equivalently.

Accelerators for neural networks require support for floating-point numbers. Furthermore, vector and matrix operations are interesting use cases for these as well. Support for Single Instruction Multiple Data (SIMD) would be an interesting extension to VADL.

The evaluated architectures are categorized as Reduced Instruction Set Architecture (RISC). Meaning, they load the values into register and instructions operate on those. Whereas, Complex Instruction Set Architecture (CISC) allows multiple memory

accesses in one instruction. The LCB does not support complex addressing modes and multiple memory accesses so far.

7.1 Missing concept for unmapped rules

In [HL10, p78], Hohenauer and Leupers presented semantic transformations because a target machine might not provide instructions for all the IR elements. For these cases, the compiler generator needs to define a semantically equivalent alternative. In their example, they transformed *neg* into a sequence of *not* and *add*. Our suggestion is to add a new pass with predefined transformations. These predefined transformations generate instruction selector patterns for unclassified instructions. However, predefined transformations are only required for operations that are not expandable by LLVM, presented in Section 3.11.

In our view, these predefined transformations could be a potentional fix for the missing remainder operation selection pattern, mentioned in Section 5.2. The missing selection pattern results into an automatic LLVM expansion that triggers a selection error because LLVM's alternative is not supported by our AArch64 VADL specification.

7.2 Multi Output Instructions

LLVM's instruction selector has only support for tree patterns. But, there are instructions with multiple outputs, e.g., the divmod instruction. The LCB cannot generate patterns for these instructions. Upstream compilers use peephole passes to detect these patterns and combine them into multiple output instructions. A possible extension for the LCB could be to automatically generate these passes.

7.3 Constant Sequences

Section 2.3.2 presented constant and register adjustment sequences for constant materialization. With the current implementation, the constant's parameter type determines the chosen constant sequence. A possible extension could enable predicates for constant and register adjustment sequences. These predicates would be evaluated during the materialization.

Additionally, a constant sequence does not explicitly define the resulting constant type in the sequence. Meaning, the constant, created by a constant sequence, will have the architecture's type. In the case of RV32, the architecture's type is a signed 32 bit integer, and for RV64 and AArch64, a signed 64 bit integer. However, more advanced ISAs could benefit from distinguishing between 32 and 64 bit constants.

Figure 7.1 shows an example of a constant sequence for RISC-V. The constant

sequence materializes a constant `imm` up to a bit-width of 12. However, the current implementation provides no information about the type of the resulting constant. As mentioned above, the LCB assumes it is the architecture's type.

```
1 constant sequence( rd : Bits<5>, imm : SInt<12> ) =
2 {
3     ADDI{ rd = rd, rs1 = 0, imm = imm }
4 }
```

Listing 7.1: Constant Sequence in VADL

CHAPTER 8

Conclusion

In this work, we reimplemented the compiler generator from scratch for OpenVADL. The reimplemented generator can generate compilers for the architectures RV32IM, RV64IM and, in limited capacity, AArch64. In contrast, the obsolete generator implementation could only generate a compiler for RV32IM and higher optimization levels were not supported. Moreover, the VADL specifications included manual adjustments to address issues with the generated compiler. The reimplemented compiler generator generates compilers that support more architectures with higher optimization levels. Unlike earlier implementations, OpenVADL's compiler generator requires no manual intervention or patches.

In comparison to upstream compilers, a compiler generated by the LCB runs 29.73% and 43.83% more instructions than upstream for RV32IM and RV64IM on average, respectively.

In Section 3.6, we presented a classification algorithm that heuristically classifies instructions based on their behavior. The classification information is then used to lower instruction behavior graphs into instruction selection patterns, as presented in Section 3.2. In particular, a method for generating instruction selector patterns is the recursive lowering approach that was presented in Section 3.7.1. Section 4 lists several problems that remain open in the current implementation. Overall, the cause of most problems is the semantic gap, described in 2.4.2. Further work is required to bridge the gap between the compiler frontend and backend, as well as between the instruction selector pattern and instruction behavior.

Overview of Generative AI Tools Used

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

List of Figures

2.1 SelectionDAG for Listing 2.5 14
2.2 Specification based ASIP design flow from [KL11, p40] 18
2.3 Overview of OpenVADL Compiler Architecture from [Fre+]. Yellow boxes represent generators. 26
2.4 VIAM behavior graph of the RISC-V ADD instruction from [Fre+] 26
2.5 Semantic Gap in OpenVADL 28

3.1 Compiler Generator Overview 31
3.2 General Lowering Approach for Machine Instructions 38

5.1 OpenVADL LCB RV32IM: Embench QEMU / relative number of executed instructions compared to upstream (lower better) 52
5.2 OpenVADL LCB RV64IM: Embench QEMU / relative number of executed instructions compared to upstream (lower better) 53

List of Tables

3.1 Properties for Instruction Classification 36

Acronyms

ABI Application Binary Interface. 19, 22, 23, 41, 43, 49, 63

ASIP Application Specific Integrated Processor. 1, 17–19, 30, 58, 71

AST Abstract Syntax Tree. 4, 24

CFG Control Flow Graph. 4, 5, 39

CISC Complex Instruction Set Architecture. 63

DAG Directed Acyclic Graph. 8, 9, 12, 13, 16

DFG Data Flow Graph. 4, 13, 39

DSE Design Space Exploration. 17, 18, 30

FastISel Fast Selection DAG Instruction Selection. 10, 13, 15

GCB Generic Compiler Backend Generator. 24, 30, 33, 36, 47, 63

GCC GNU Compiler Collection. 8, 30, 51

GlobalISel Global Selection Dag Instruction Selection. 10, 12, 15, 16

gMIR Generic Machine Intermediate Representation. 15, 16, 27

HDL Hardware Description Language. 17–19

IR Intermediate Representation. 3–9, 12, 13, 15, 16, 24, 27, 30, 43, 52, 57–60, 64

ISA Instruction Set Architecture. 3, 7, 13, 19, 23, 36, 41, 45, 57–59, 63, 64

ISS Instruction Set Simulator. 19, 24

JIT Just-In-Time. 30

LCB LLVM Compiler Backend Generator. 24, 30, 36, 39–43, 45–49, 51–53, 57–60, 63–65, 67

LLVM LLVM. 3, 6, 9, 10, 12, 13, 15, 16, 24, 30, 33, 34, 36, 38, 39, 41, 43, 46, 48, 54, 57–59, 64

LLVM IR LLVM Intermediate Representation. 3, 6, 10, 12, 13, 15, 16, 27, 28, 37, 40, 41, 43, 45–47, 60

MIA Micro Architecture. 19, 58

MIR Machine Intermediate Representation. 16

MLIR Multi Level Intermediate Representation. 5

PC Program Counter. 21, 22, 29, 33–36, 42

PDG Program Dependence Graph. 5

PDL Processor Description Language. 1, 17, 57, 58, 60

RISC Reduced Instruction Set Architecture. 63

SDISel Selection DAG Instruction Selection. 10, 12, 13, 15, 16, 40

SDNode Selection Directed Acyclic Graph Node. 12, 39, 43–45, 48

SelectionDAG Selection DAG Instruction Selection's Directed Acyclic Graph. 10, 12, 13, 27, 39, 46

SIMD Single Instruction Multiple Data. 63

SSA Static Single Assignment Form. 4, 6, 16, 41

TableGen TableGen. 9, 10, 12, 30, 34, 37, 38, 49, 58, 60

VADL Vienna Architecture Description Language. 1, 3, 10, 19, 21–24, 26–31, 33, 37, 39–49, 57, 59, 63–65, 67

VIAM VADL Intermediate Architecture Model. 24, 27, 29, 30

Bibliography

[ACB12] Rafael Auler, Paulo Cesar Centoducatte, and Edson Borin. "ACCGen: An Automatic ArchC Compiler Generator". In: *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing. ISSN: 1550-6533. Oct. 2012, pp. 278–285. DOI: 10.1109/SBAC-PAD.2012.33. (Visited on 15. 09. 2025).

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. "Code generation using tree matching and dynamic programming". In: *ACM Transactions on Programming Languages and Systems* 11.4 (Oct. 1989), pp. 491–516. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/69558.75700. (Visited on 20. 03. 2024).

[Anoa] Anonymous. *Core Pipeline — LLVM*. URL: https://llvm.org/docs/GlobalISel/Pipeline.html (visited on 24. 09. 2025).

[Anob] Anonymous. *Embench*. A Modern Embedded Benchmark Suite. URL: https://www.embench.org/ (visited on 04. 11. 2025).

[Anoc] Anonymous. *How to handle truncation when subregister? - Code Generation*. LLVM Discussion Forums. URL: https://discourse.llvm.org/t/how-to-handle-truncation-when-subregister/88497/4 (visited on 01. 11. 2025).

[Anod] Anonymous. *Improve Instruction Selection - GCC Wiki*. URL: https://gcc.gnu.org/wiki/Improve_instruction_selection (visited on 03. 11. 2025).

[Anoe] Anonymous. *InstructionSelect — LLVM*. URL: https://llvm.org/docs/GlobalISel/InstructionSelect.html (visited on 24. 09. 2025).

[Anof] Anonymous. *IRTranslator — LLVM*. URL: https://llvm.org/docs/GlobalISel/IRTranslator.html#irtranslator (visited on 24. 09. 2025).

[Anog] Anonymous. *Legalizer — LLVM*. URL: https://llvm.org/docs/GlobalISel/Legalizer.html (visited on 24. 09. 2025).

[Anoh] Anonymous. *LLVM IR Undefined Behavior (UB) Manual — LLVM 22.0.0git documentation*. URL: https://llvm.org/docs/UndefinedBehavior.html (visited on 27. 10. 2025).

[Anoi] Anonymous. *LLVM Language Reference Manual — LLVM 22.0.0git documentation*. URL: https://llvm.org/docs/LangRef.html (visited on 26. 10. 2025).

[Anoj] Anonymous. *LLVM: Scalar evolution*. URL: https://www.npopov.com/2023/10/03/LLVM-Scalar-evolution.html (visited on 26. 10. 2025).

[Anok] Anonymous. *LLVM's Analysis and Transform Passes — LLVM 22.0.0git documentation*. URL: https://llvm.org/docs/Passes.html#scalar-evolution-scalar-evolution-analysis (visited on 26. 10. 2025).

[Anol] Anonymous. *RegBankSelect — LLVM 22.0.0git documentation*. URL: https://llvm.org/docs/GlobalISel/RegBankSelect.html (visited on 24. 09. 2025).

[Anom] Anonymous. *The LLVM Target-Independent Code Generator*. URL: https://llvm.org/docs/CodeGenerator.html#built-in-register-allocators (visited on 24. 09. 2025).

[Anon] Anonymous. *The LLVM Target-Independent Code Generator — LLVM 22.0.0git documentation*. URL: https://llvm.org/docs/CodeGenerator.html#selectiondag-legalize-phase (visited on 09. 11. 2025).

[Ano24] Anonymous. *TableGen LLVM Documentation*. 29th Mar. 2024. URL: https://llvm.org/docs/TableGen/ (visited on 22. 03. 2024).

[Ano25a] Anonymous. *Glue for conditional instructions - Code Generation / AArch64*. LLVM Discussion Forums. Section: Code Generation. 22nd Aug. 2025. URL: https://discourse.llvm.org/t/glue-for-conditional-instructions/79356/5 (visited on 26. 10. 2025).

[Ano25b] Anonymous. *RISC-V ELF Abi*. 9th Dec. 2025. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc (visited on 12. 09. 2025).

[BEK07] Florian Brandner, Dietmar Ebner, and Andreas Krall. "Compiler generation from structural architecture descriptions". In: *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. ESWEEK07: Third Embedded Systems Week. Salzburg Austria: ACM, 30th Sept. 2007, pp. 13–22. ISBN: 978-1-59593-826-8. DOI: 10.1145/1289881.1289886. (Visited on 28. 03. 2024).

[Bel05] Fabrice Bellard. “QEMU, a fast and portable dynamic translator”. In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC ’05. USA: USENIX Association, 10th Apr. 2005, p. 41. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf (visited on 19. 11. 2025).

[Ben] Eli Bendersky. *A deeper look into the LLVM code generator, Part 1*. URL: https://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1 (visited on 23. 09. 2025).

[Ber+25] Ryan Berger, Mitch Briles, Nader Boushehrinejad Moradi, Nicholas Coughlin, Kait Lam, Nuno P. Lopes, Stefan Mada, Tanmay Tirpankar, and John Regehr. “Translation Validation for LLVM’s AArch64 Backend”. In: *Proceedings of the ACM on Programming Languages* 9 (OOPSLA2 9th Oct. 2025), pp. 2710–2735. ISSN: 2475-1421. DOI: 10.1145/3763147. (Visited on 22. 10. 2025).

[BPK13] Florian Brandner, Viktor Pavlu, and Andreas Krall. “Automatic generation of compiler backends”. In: *Software: Practice and Experience* 43.2 (2013), pp. 207–240. ISSN: 1097-024X. DOI: 10.1002/spe.2106. (Visited on 14. 11. 2025).

[Bra09] Florian Brandner. “Compiler Backend Generation from Structural Processor Models”. PhD thesis. Technische Universität Wien, 2009. 165 pp. DOI: 20.500.12708/12553.

[Bra10] Florian Brandner. “Completeness of automatically generated instruction selectors”. In: *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors. ISSN: 1063-6862. July 2010, pp. 175–182. DOI: 10.1109/ASAP.2010.5540994. (Visited on 23. 10. 2025).

[Cen+04] Jianjiang Ceng, Weihua Sheng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. “Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting”. In: *Computer Systems: Architectures, Modeling, and Simulation*. Ed. by Andy D. Pimentel and Stamatis Vassiliadis. Berlin, Heidelberg: Springer, 2004, pp. 463–473. ISBN: 978-3-540-27776-7. DOI: 10.1007/978-3-540-27776-7_48.

[Cen+05a] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. “C Compiler Retargeting Based on Instruction Semantics Models”. In: *Design, Automation and Test in Europe*. Design, Automation and Test in Europe. ISSN: 1558-1101. Mar. 2005, pp. 1150–1155. DOI: 10.1109/DATE.2005.88. (Visited on 25. 08. 2024).

[Cen+05b] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. “C Compiler Retargeting Based on Instruction Semantics Models”. In: *Design, Automation and Test in Europe*. Design, Automation and Test in Europe. ISSN: 1558-1101. Mar. 2005, pp. 1150–1155. DOI: 10.1109/DATE.2005.88. (Visited on 15. 09. 2025).

[Cha82] G. J. Chaitin. “Register allocation & spilling via graph coloring”. In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction - SIGPLAN ’82*. the 1982 SIGPLAN symposium. Boston, Massachusetts, United States: ACM Press, 1982, pp. 98–101. ISBN: 978-0-89791-074-3. DOI: 10.1145/800230.806984. (Visited on 27. 04. 2025).

[Col25] Quentin Colombet. *LLVM code generation: a deep dive into compiler backend development*. 1st ed. Birmingham: Packt Publishing, Limited, 2025. ISBN: 978-1-83546-257-7. URL: https://www.packtpub.com/en-us/product/llvm-code-generation-9781835462577.

[Coo12] Keith Cooper. *Engineering a compiler*. In collab. with Linda Torczon. 2nd ed. Amsterdam: Elsevier, 2012. 825 pp. ISBN: 978-1-282-97670-2.

[DF04] Jack W. Davidson and Christopher W. Fraser. “Automatic generation of peephole optimizations”. In: *ACM SIGPLAN Notices* 39.4 (Apr. 2004), pp. 104–111. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/989393.989407. (Visited on 24. 10. 2025).

[DF84] Jack W. Davidson and Christopher W. Fraser. “Code selection through object code optimization”. In: *ACM Transactions on Programming Languages and Systems* 6.4 (Oct. 1984), pp. 505–526. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/1780.1783. (Visited on 23. 09. 2025).

[Dob13] Jeroen Dobbelaere. “Automatic Generation of LLVM Backends from LISA Using Processor Designer”. LLVM Developer Meeting. San Francisco, CA, USA, 2013. URL: https://llvm.org/devmtg/2013-11/slides/Dobbelaere-Poster.pdf.

[Ebn+08] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. “Generalized instruction selection using *SSA* -graphs”. In: *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. LCTES08: ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems. Tucson AZ USA: ACM, 12th June 2008, pp. 31–40. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375663. (Visited on 28. 10. 2025).

[Ert99] M. Anton Ertl. “Optimal code selection in DAGs”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL99: Symposium on Prinicples of Programming Languages 1999. San Antonio Texas USA: ACM, Jan. 1999, pp. 242–249. ISBN: 978-1-58113-095-9. DOI: 10.1145/292540.292562. (Visited on 20. 03. 2024).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/24039.24041. (Visited on 23. 10. 2025).

[Fre+] 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*. URL: https://arxiv.org/pdf/2402.09087v2.

[Fre+26] Florian Freitag, Linus Halder, Benedikt Huber, Benjamin Kasper, Michael Nestler, Kevin Per, Matthias Raschhofer, Alexander Ripar, Johannes Zottele, and Andreas Krall. “OpenVADL: An Open Source Implementation of the Vienna Architecture Description Language”. In: *Architecture of Computing Systems*. Cham: Springer Nature Switzerland, 2026, pp. 156–171. ISBN: 978-3-032-03281-2. DOI: 10.1007/978-3-032-03281-2_11.

[GM04] Tilman Glökler and Heinrich Meyr. *Design of Energy-Efficient Application-Specific Instruction Set Processors*. Boston: Kluwer Academic Publishers, 2004. ISBN: 978-1-4020-7730-2. DOI: 10.1007/b105292. (Visited on 14. 09. 2025).

[Gra21] Alexander Graf. “Compiler Backend Generation using the VADL Processor Description Language”. Master’s Thesis. Technische Universität Wien, 13th Apr. 2021. DOI: 10.34726/hss.2021.79221.

[Hjo16] Gabriel Hjort Blindell. *Instruction Selection*. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-34017-3 978-3-319-34019-7. DOI: 10.1007/978-3-319-34019-7. (Visited on 22. 03. 2024).

[HKS03] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. “Graph coloring vs. optimal register allocation for optimizing compilers”. In: Modular Programming Languages. Book Title: Lecture notes in computer science ISSN: 0302-9743. Berlin: Springer, 2003, pp. 202–213. ISBN: 978-3-540-40796-6. DOI: 10.1007/978-3-540-45213-3_26.

[HL10] Manuel Hohenauer and Rainer Leupers. *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. 1. Aufl. New York, NY: Springer-Verlag, 2010. ISBN: 978-1-4419-1175-9. DOI: 10.1007/978-1-4419-1176-6.

[HML02] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with LISA*. Boston, MA: Springer US, 2002. ISBN: 978-1-4419-5334-6 978-1-4757-4538-2. DOI: 10.1007/978-1-4757-4538-2. (Visited on 15. 09. 2025).

[Hoh+04] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. “A methodology and tool suite for C compiler generation from ADL processor models”. In: *Automation and Test in Europe Conference and Exhibition Proceedings Design*. Automation and Test in Europe Conference and Exhibition Proceedings Design. Vol. 2. ISSN: 1530-1591. Feb. 2004, 1276–1281 Vol.2. DOI: 10.1109/DATE.2004.1269071. (Visited on 25. 08. 2024).

[Hsu24] Min-Yih Hsu. *Legalizations in LLVM Backend*. Min Hsu’s Homepage. Section: blog. 15th May 2024. URL: https://myhsu.xyz/llvm-codegen-legalization/ (visited on 12. 09. 2025).

[Hsu25] Min-Yih Hsu. *Machine Scheduler in LLVM - Part I*. Min Hsu’s Homepage. Section: blog. 16th Sept. 2025. URL: https://myhsu.xyz/llvm-machine-scheduler/ (visited on 30. 09. 2025).

[Inc] Apple Inc. *Swift ABI Stability Manifesto*. GitHub. URL: https://github.com/swiftlang/swift/blob/main/docs/ABIStabilityManifesto.md (visited on 24. 10. 2025).

[Kar+08] Kingshuk Karuri, Anupam Chattopadhyay, Xiaolin Chen, David Kammler, Ling Hao, Rainer Leupers, Heinrich Meyr, and Gerd Ascheid. “A Design Flow for Architecture Exploration and Implementation of Partially Reconfigurable Processors”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.10 (Oct. 2008), pp. 1281–1294. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2008.2002685. (Visited on 06. 09. 2025).

[KL11] Kingshuk Karuri and Rainer Leupers. *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*. New York, NY: Springer New York, 2011. ISBN: 978-1-4419-8254-4 978-1-4419-8255-1. DOI: 10.1007/978-1-4419-8255-1. (Visited on 14. 09. 2025).

[Kuc+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. “Dependence graphs and compiler optimizations”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’81*. the 8th ACM SIGPLAN-SIGACT symposium. Williamsburg, Virginia: ACM Press, 1981, pp. 207–218. ISBN: 978-0-89791-029-3. DOI: 10.1145/567532.567555. (Visited on 24. 10. 2025).

[Lat+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul, Korea (South): IEEE, 27th Feb. 2021, pp. 2–14. ISBN: 978-1-7281-8613-9. DOI: 10.1109/CGO51591.2021.9370308. (Visited on 30. 03. 2024).

[Lez14] Roland Lezuo. “Scalable translation validation: tools, techniques and framework”. Thesis. Technische Universität Wien, 2014. DOI: 10.34726/hss.2014.24883. (Visited on 04. 11. 2025).

[Muc06] Steven S. Muchnick. *Advanced compiler design and implementation*. San Francisco, Calif. [u.a.]: Kaufmann, 2006. xxix+856. ISBN: 978-1-55860-320-2. URL: https://dl.acm.org/doi/book/10.5555/286076.

[Ole11] Jakob Stoklund Olesen. *Greedy Register Allocation in LLVM 3.0*. The LLVM Project Blog. Section: posts. 18th Sept. 2011. URL: https://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html (visited on 25. 09. 2025).

[Pee+99] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. “LISA—machine description language for cycle-accurate models of programmable DSP architectures”. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. DAC99: The 36th ACM/IEEE-CAS/EDAC Design Automation Conference. New Orleans Louisiana USA: ACM, June 1999, pp. 933–938. ISBN: 978-1-58113-109-3. DOI: 10.1145/309847.310101. (Visited on 28. 03. 2024).

[Pop25] Nikita Popov. *[RFC] An ABI lowering library for LLVM - LLVM Project*. LLVM Discussion Forums. Section: LLVM Project. 7th Feb. 2025. URL: https://discourse.llvm.org/t/rfc-an-abi-lowering-library-for-llvm/84495 (visited on 29. 09. 2025).

[PS99] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation”. In: *ACM Transactions on Programming Languages and Systems* 21.5 (Sept. 1999), pp. 895–913. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/330249.330250. (Visited on 27. 04. 2025).

[RB22] Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based Compiler Design*. 1st ed. 2022. Cham: Springer International Publishing Imprint: Springer, 2022. ISBN: 978-3-030-80515-9. DOI: 10.1007/978-3-030-80515-9. (Visited on 22. 09. 2025).

[Rig+04] 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*. 16th Symposium on Computer Architecture and High Performance Computing. ISSN: 1550-6533. Oct. 2004, pp. 66–73. DOI: 10.1109/SBAC-PAD.2004.8. (Visited on 06. 09. 2025).

[Sch+07] Hanno Scharwaechter, David Kammler, Andreas Wieferink, Manuel Hohenauer, Kingshuk Karuri, Jianjiang Ceng, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. “ASIP architecture exploration for efficient IPSec encryption: A case study”. In: *ACM Transactions on Embedded Computing Systems* 6.2 (May 2007), p. 12. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/1234675.1234679. (Visited on 06. 09. 2025).

[TEK18] Patrick Thier, M. Anton Ertl, and Andreas Krall. “Fast and flexible instruction selection with constraints”. In: *Proceedings of the 27th International Conference on Compiler Construction*. CGO ’18: 16th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Vienna Austria: ACM, 24th Feb. 2018, pp. 93–103. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179501. (Visited on 20. 03. 2024).

[WP94] P.G. Whiting and R.S.V. Pascoe. “A history of data-flow languages”. In: *IEEE Annals of the History of Computing* 16.4 (1994), pp. 38–59. ISSN: 1934-1547. DOI: 10.1109/85.329757. (Visited on 25. 10. 2025).

[Zho+25] Ming Zhong, Fang Lv, Lulin Wang, Lei Qiu, Yingying Wang, Ying Liu, Huimin Cui, Xiaobing Feng, and Jingling Xue. “VEGA: Automatically Generating Compiler Backends using a Pre-trained Transformer Model”. In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO ’25: 23rd ACM/IEEE International Symposium on Code Generation and Optimization. Las Vegas NV USA: ACM, Mar. 2025, pp. 90–106. ISBN: 979-8-4007-1275-3. DOI: 10.1145/3696443.3708931. (Visited on 19. 08. 2025).