TU WIEN Informatics

Extending the OpenVADL compiler generator

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Informatik

eingereicht von

Tobias Weiss
Matrikelnummer 12321994

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 9. September 2026

Tobias Weiss	Andreas Krall

TU WIEN Informatics

Extending the OpenVADL compiler generator

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Informatics

by

Tobias Weiss
Registration Number 12321994

to the Faculty of Informatics

at the TU Wien

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

Vienna, September 9, 2026

Tobias Weiss	Andreas Krall

Erklärung zur Verfassung der Arbeit

Tobias Weiss

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, 9. September 2026

Tobias Weiss

Danksagung

Ich möchte Prof. Krall dafür danken, dass er mir die Teilnahme am OpenVADL-Projekt ermöglicht hat, sowie für seine Betreuung und kontinuierliche Unterstützung während meiner Arbeit an OpenVADL und für sein ausführliches und zeitnahes Feedback zu dieser Thesis. Darüber hinaus danke ich Kevin Per für seine technische Unterstützung und die Reviews meines Codes. Außerdem danke ich Johannes Gisy, der mir ebenfalls technische Unterstützung beim Einstieg in OpenVADL geleistet hat.

Letztlich möchte ich meiner Familie und besonders meinem Vater Ing. Alexander Weiss danken.

Acknowledgements

I want to thank Prof. Krall for allowing me to be part of the OpenVADL project, for his guidance and continuous support while working on OpenVADL, and for his detailed and timely feedback regarding this thesis. Furthermore, I thank Kevin Per for his technical support and code reviews. Additionally, I thank Johannes Gisy, who too provided technical support while getting started on OpenVADL.

Finally, I want to thank my family, and especially my father, Ing. Alexander Weiss.

Kurzfassung

OpenVADL ist eine Processor Description Language (PDL), welche anhand einer einzelnen Spezifikation einen Compiler, Linker, Assembler, Instruction Set Simulator und ähnliche Artefakte produziert. Dabei wird der Aufwand, der mit der Implementierung und Instandhaltung jedes einzelnen Teils einer kompletten Compiler-Suite verbunden ist, vermieden. In OpenVADL ist das LLVM Compiler Backend (LCB) zuständig für die Generierung des Compilers, wobei hier LLVM mit, für die Zielarchitektur spezifischen, Komponenten erweitert wird, welche von der Spezifikation abgeleitet werden.

Diese Arbeit identifiziert und behebt vier Probleme im LCB, welche beim Generieren und Verwenden des abgeleiteten Compilers für die Spezifikation eines Subsets der AArch64-Architektur zum Vorschein gekommen sind. Dabei handelt es sich um Probleme bei der Darstellung der Register-Subregister-Hierarchie, Inkonsistenzen in der Darstellung von Typen von Immediate-Operanden in OpenVADL und LLVM, die Generierung von überschüssigen Prädikatsfunktionen, welche ein internes Limit von LLVM überschritten haben, und fehlende Klassifizierungen von Instruktionen.

Die Verbesserungen dieser Arbeit werden mit einer Kollektion von C-Testdateien evaluiert, wobei jede mit dem generierten Compiler vor und nach der jeweiligen Behebung kompiliert und ausgeführt wird. Ein Großteil der Testdateien, welche vor dieser Arbeit nicht kompilierbar waren, bzw. fehlerhaft ausgeführt wurden, ist nun kompilierbar und wird korrekt ausgeführt, wobei eine kleine Anzahl an offenen Problemen noch besteht.

Abstract

OpenVADL is a Processor Description Language (PDL) that generates compilers, linkers, assemblers, instruction set simulators, and other artifacts from a single architecture specification, avoiding the effort of separately building and maintaining each tool of a compiler suite. In OpenVADL, the LLVM Compiler Backend (LCB) is responsible for producing a compiler by extending LLVM with target-specific components derived from the specification.

This thesis identifies and fixes four shortcomings in the LCB that became apparent while generating a compiler for a specification covering a subset of the AArch64 architecture. The defects concern the handling of the register and subregister hierarchy, a type mismatch between immediate operands in OpenVADL and LLVM, the generation of redundant predicate functions exceeding an internal limit of LLVM's instruction selector, and lacking classification of machine instructions.

The improvements are evaluated with a suite of C test files, each compiled with the generated compiler before and after the respective fix and executed to verify correctness. Most test files that previously failed to compile or produced incorrect results now compile and execute correctly, while a small number of open issues remain.

Contents

CHAPTER 1

Introduction

Building a compiler suite for an architecture requires immense work, as a compiler, linker, and assembler have to be implemented [ALSU06]. This is especially true when the target architecture is still in development. Existing frameworks, like LLVM [LA04], aid in the creation of a compiler and associated tools, but require cumbersome and large specifications. For effective design space exploration, one additionally wants to make use of instruction set simulators. Developing, maintaining, and keeping all of these tools in sync requires great effort.

OpenVADL [FHH+25b] is an open-source implementation of the PDL Vienna Architecture Description Language (VADL) [FHH+25a], which enables the definition of an architecture in a single specification, allowing the automatic generation of compilers, linkers, assemblers, instruction set simulators, and other artifacts [FHH+25b] [CML08]. This work extends the compiler generator of OpenVADL, addressing shortcomings that became apparent while working with a specification for a subset of the AArch64 architecture.

Chapter 2 introduces necessary background information on LLVM, OpenVADL, and AArch64, and discusses related work, while chapter 3 describes changes made to the compiler generator, which are further evaluated in chapter 4. Topics for future work are discussed in chapter 5. Finally, chapter 6 concludes this thesis.

CHAPTER 2

Background and Related Work

This chapter introduces necessary background information on LLVM, the AArch64 architecture, VADL, and OpenVADL. Furthermore, related work is discussed in section 2.4.

2.1 LLVM

LLVM [LA04] is a collection of largely target-independent components, which cover the tasks essential to a compiler backend, including instruction selection, instruction scheduling, and register allocation. LLVM's supported architectures can be extended by describing a target architecture in terms of its available instructions, registers, calling conventions, and other necessary details. [LA04] [Prob]

A target architecture is described declaratively in LLVM's domain-specific language, TableGen, and, in addition, C++ implementations must be provided for transformations that cannot be expressed declaratively, such as custom instruction lowering [LA04] [Prok].

Source programs are transformed into LLVM Intermediate Representation (LLVM-IR), handled by LLVM's target-independent components, and ultimately compiled into target-dependent instructions [LA04] [Prod].

2.1.1 LLVM Intermediate Representation

LLVM-IR is a well-defined, Static Single Assignment (SSA) [RT22] based intermediate representation used by LLVM as an internal representation of source code within the compiler, and is designed to be expressive enough to allow the representation of high-level language features. Optimizations [Muc98], analyses, and transformations are directly done on the LLVM-IR. LLVM-IR comes in three different, equivalent forms: an in-memory representation, an on-disk bitcode representation, and human-readable assembly language. [Prod] [LA04]

Listing 2.1 shows an example of LLVM-IR in its human-readable assembly language form.

```
1 define i64 @square(i32 %x) {
2   %1 = sext i32 %x to i64
3   %2 = mul i64 %1, %1
4   ret i64 %2
5 }
```

Listing 2.1: Example of a function squaring its input, written in LLVM-IR.

Listing 2.1 defines a function named `square` with a single input parameter of type `i32`, representing a 32-bit integer, and a return value of type `i64`. The function uses the `sext` instruction to sign-extend the 32-bit parameter to 64 bits, then the `mul` instruction to multiply the extended value by itself, before returning the result.

Zero-, Sign- and Any-Extension

LLVM-IR is strongly and statically typed. Integer types of different widths are distinct types, e.g., `i32` and `i64` cannot be mixed directly in an operation [Prod]. Therefore, a narrower value must first be explicitly converted to match a wider one, as can be seen in the previous section's example 2.1.

Two instructions perform this widening at the LLVM-IR level. `sext` sign-extends a value, replicating its sign bit into the newly added upper bits and thus preserving the value it represents as a signed integer. `zext` zero-extends a value instead, filling the new upper bits with zero and thus preserving the value it represents as an unsigned integer. The two differ only in how they fill those upper bits, and therefore only produce different results for negative inputs. [Prod]

2.1.2 TableGen

TableGen is a domain-specific language extensively used in LLVM to describe necessary information about a target architecture. This includes available registers, instructions, instruction selection patterns, and other necessary details [Proj] [Prok].

A TableGen file consists of two major parts: *definitions* and *classes*, which are both considered *records* in TableGen's jargon. Records are identified by a unique name and hold domain-specific information for the target in the form of associated values [Proj].

Definitions are concrete instantiations of *records*, which may make use of *classes*, which themselves are abstract *records* and allow developers to build abstractions.

TableGen files themselves have no real meaning, and so-called backends are used to interpret a given TableGen file in a specific context and generate backend-dependent output. This output may range from C header include files to textual descriptions of the domain-specific information. [Proj]

Listing 2.2 shows a concrete example of the TableGen class `Register` and an associated definition of a register `SP`, using this class.

```
class Register<string n> {
  string Namespace = "";
  string AsmName = n;
  bits<16> HWEncoding = 0;
  bit isArtificial = false;
}

def SP : Register<"SP">
{
    let Namespace = "aarch64";
    let HWEncoding = 31;
    let isArtificial = 1;
}
```

Listing 2.2: Example of a TableGen class `Register` and a concrete instantiation `SP`.

2.1.3 SelectionDAG

SelectionDAG is a framework used for instruction selection. LLVM-IR is transformed into a separate intermediate representation, namely a Directed Acyclic Graph (DAG), which is then gradually transformed while moving through a pipeline of transformations. Figure 2.1 shows this pipeline and individual transformations. [Proa]

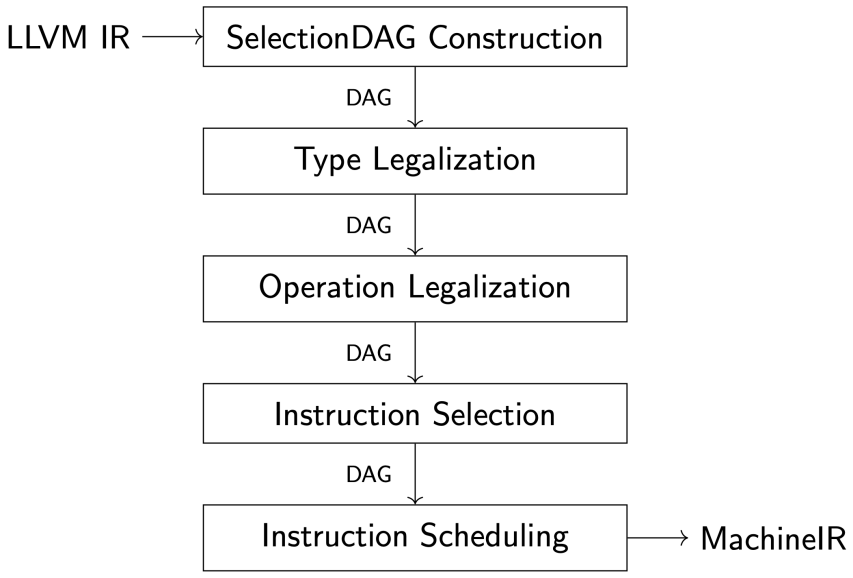

Figure 2.1: SelectionDAG's pipeline of transformations, taking LLVM-IR as input.

SelectionDAG's DAG representation is built upon individual `SDNodes`, where each represents a single node in the DAG. Each `SDNode` carries an associated opcode, which represents an operation to be performed, including addition, branching, or subtraction,

and additional flags and constants. A `SDNode` can have outputs and inputs. The outputs are represented by `SDValue` instances, which refer to the `SDNode` producing the value. Inputs are represented by `SDUse` instances, which reference the `SDValue` being used and a `SDNode` that is the user of this value. [Proa]

These references between `SDNodes`, `SDValues`, and `SDUses` are represented by edges in the DAG. Figure 2.2 shows a simplified example of a SelectionDAG generated for the LLVM-IR of listing 2.3. Each node represents a `SDNode` instance, where `ISD::` denotes the associated operation, the numbered boxes inside a node represent the inputs of the node, `SDUses`, and `MVT::` denotes the output of the node, `SDValues`. [Proa]

```
1 %c = add i32 2, 3
2 %d = mul i32 %b, %c
```

Listing 2.3: Example of two LLVM-IR instructions, which add two constants and multiply a variable `%b` with the result of the addition.

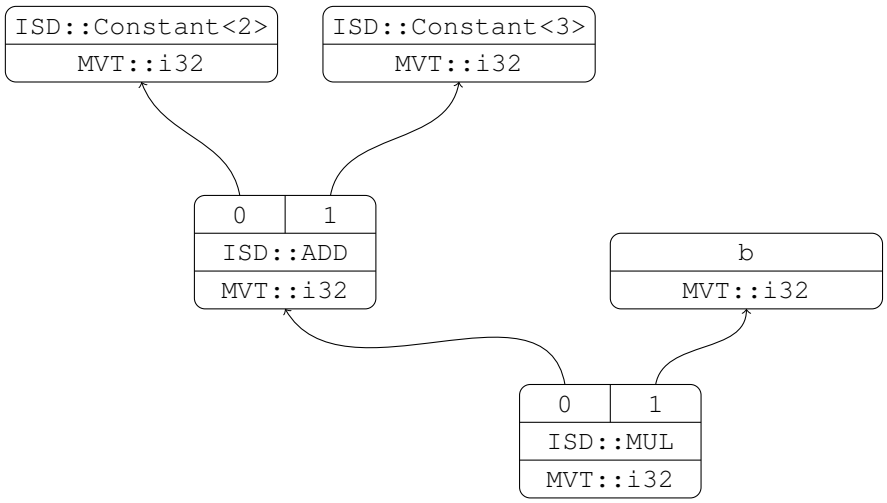

Figure 2.2: Simplified example of a SelectionDAG representing the LLVM-IR of listing 2.3.

Instructions

Instructions for a target architecture are defined as TableGen records, as mentioned in section 2.1.2. An instruction is comprised of information about the encoded size of the instruction, input operands, output operands, flags describing the behavior of the instruction, and other details. [Proj]

A simplified example of a variant of AArch64's `add` instruction can be seen in listing 2.4. `OutOperandList` describes the result produced by the instruction, which in this case is a value in a register of the `X` register class, which is further described in section 2.3. `InOperandList` describes input operands of the instruction, in this case a register of the register class `X` and an immediate. Immediates are further described in section 2.1.3.

```
def ADDXI : Instruction
{
let Namespace = "aarch64";
let Size = 4;

let OutOperandList = ( outs X:$rd );
let InOperandList = ( ins X:$rn, AArch64Base_ADDXI_imm12XAsInt64:
    $imm12X );

let isTerminator       = 0;
let isBranch           = 0;
let isCall             = 0;
let isReturn           = 0;
let isPseudo           = 0;
}
```

Listing 2.4: Simplified example of a variant of AArch64's `add` instruction.

Immediates

Similar to instructions, immediates are defined as TableGen records and represent immediate values, which may be used in instruction input or output operands. Immediates have an associated encode and decode method, which is implemented as C++ functions handling the encoding and decoding of immediates as present in a decoded or encoded instruction. [Proj]

Furthermore, immediates are generally only valid for a given number domain, which is enforced by a predicate function, which is implemented in C++. The associated predicate function takes the immediate as input and rejects it if it is outside the valid domain of the given instruction.

Listing 2.5 shows a TableGen class and definition for the immediate mentioned in listing 2.4 of the previous section, which represents a 12-bit unsigned integer. The class definition specifies the names of the necessary encoding and decoding functions. The concrete record instantiation beneath actually defines the immediate, extending the class above and further specifying a predicate function, which is responsible for enforcing the immediate's validity.

```
class ADDXI_imm12X<ValueType ty> : Operand<ty>
{
  let EncoderMethod = "ADDXI_wrapper";
  let DecoderMethod = "ADDXI_imm12X_decode_wrapper";
}

def ADDXI_imm12XAsInt64
    : ADDXI_imm12X<i64>
    , ImmLeaf<i64, [{ return Imm12X_predicate(Imm); }]>;
```

Listing 2.5: Example of a 12-bit immediate as defined in TableGen.

Registers

As mentioned in section 2.1.2, registers for a given target architecture are defined as TableGen records. A simplified example can be seen in listing 2.2.

LLVM expects a target to define register classes, which define the size of registers included in that class and information about alignment and the width of registers. Furthermore, a list of individual register definitions has to be provided, describing the registers part of that class. [Proj]

An individual register definition includes information about the name of the register, available subregisters, and other necessary information. As each register defines available subregisters, a hierarchy of registers is implicitly defined. Each specified subregister additionally has to have an associated subregister-index, which denotes a name that can be used to access a given subregister from a register. [Proj]

Listing 2.6 shows an excerpt of the generated register classes and registers for the AArch64 OpenVADL specification. Lines 1-4 define the register class `X`, a register class available in the AArch64 architecture, as described in section 2.3. The definition includes a list of registers belonging to that given register class.

Line 11 defines the necessary subregister index, which is further used by the definition of register `X0` to specify that the narrower register `W0`, defined on line 20, is a subregister.

```
1  def X : RegisterClass<
2    ( add X0, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13,
       X14, X15, X16, X17, X18, X19, X20, X21, X22, X23, X24, X25, X26,
       X27, X28, X29, X30, X31 )
3  > {
4  }
5
6  def W : RegisterClass<
7    ( add W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, W10, W11, W12, W13,
       W14, W15, W16, W17, W18, W19, W20, W21, W22, W23, W24, W25, W26,
       W27, W28, W29, W30, W31 )
8  > {
9  }
10
11 def SUB_32  : SubRegIndex<32>;
12
13 def X0 : Register<"X0">
14 {
15     let Namespace = "aarch64";
16     let SubRegs = [ W0 ];
17     let SubRegIndices = [ SUB_32 ];
18 }
19
20 def W0 : Register<"W0">
21 {
22     let Namespace = "aarch64";
```

```
23      let Aliases = [ ];
24      let SubRegs = [ ];
25      let SubRegIndices = [ ];
26  }
```

Listing 2.6: Simplified example of two register class and two register definitions for the AArch64 architecture.

Instruction-Selection Patterns

SelectionDAG selects appropriate instructions by matching DAG nodes and subgraphs against a list of patterns, which describe how a given subgraph may be represented in the target architecture's instruction set. A pattern is a TableGen record, which consists of two parts: a DAG of target-independent `SDNode` opcodes and a DAG of target-dependent instructions.

SelectionDAG matches a given DAG against patterns, and if a matching pattern is found, the DAG is replaced by the target-dependent instructions as specified in the pattern.

Listing 2.7 shows an example of a pattern for the `add` instruction shown in section 2.1.3. The `add` represents the `SDNode`'s target-independent opcode; `ADDXI` represents AArch64's add instruction.

```
1  def : Pat<(add X:$rn, AArch64Base_ADDXI_imm12XAsInt64:$imm12X),
2          (ADDXI X:$rn, AArch64Base_ADDXI_imm12XAsInt64:$imm12X)>;
```

Listing 2.7: Example of a SelectionDAG pattern for AArch64's `add` instruction

2.2 OpenVADL and VADL

OpenVADL is an open-source implementation of the PDL VADL [FHH+25b] [FHH+25a]. VADL allows the specification of a processor's architecture, including its Instruction Set Architecture (ISA) and Microarchitecture (MiA). Given these specifications, OpenVADL is able to, among other things, generate a compiler, linker, and assembler for the described architecture, functional and cycle-accurate instruction set simulators, and test cases [FHH+25b] [CML08].

An OpenVADL specification generally consists of an ISA, which is used by the compiler generator together with one or more Application Binary Interface (ABI) descriptions. Figure 2.3 shows the internal pipeline of OpenVADL and how a specification is transformed and used to generate various artefacts.

The compiler artefact shown in figure 2.3 is further split into the Generic Compiler Backend (GCB) and LCB, where the former handles necessary transformations and interpolations for any compiler backend, while the latter focuses on LLVM-specific transformations and file generation [FHH+25b].

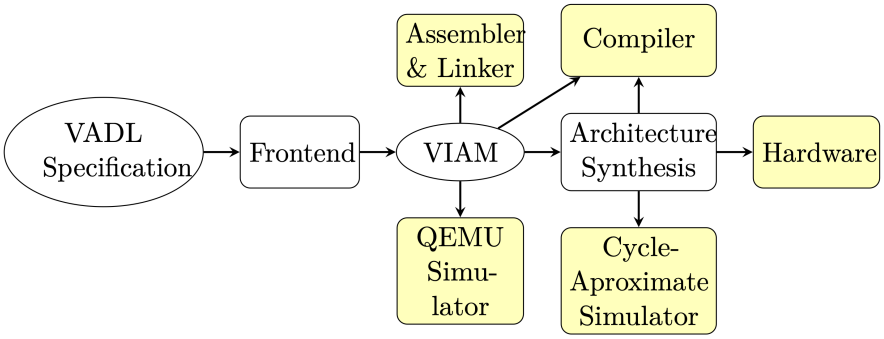

Figure 2.3: OpenVADL pipeline, generating several artefacts, highlighted in yellow, from a given input specification. Taken from [FHH+25b]

2.2.1 Semantic Gap

As described by Per [Per25], there is a semantic gap between OpenVADL and LLVM, which stems from the fact that LLVM requires a target to be specified in a specific structure. This structure has to be derived and interpolated from the available information present in a VADL specification. The semantic gap describes the differences or mismatches between the semantics of a VADL specification and LLVM.

To overcome this semantic gap, OpenVADL performs instruction classification, which categorises instructions based on their behavior [Per25]. Table 2.1 shows a small number of classification labels and associated properties, which must be met by an instruction to be classified with that label.

Classification Label	**Properties for Classification**
ADD	Builtin *vadl::ADD* and two register file operands
XOR	Builtin *vadl::XOR* and one register file operand and an immediate operand
SLTH	Builtin *vadl::SLTH* and two register file operands

Table 2.1: Properties for Instruction Classification. This table is an excerpt taken from [Per25].

2.2.2 Instructions

Instruction definitions are part of the ISA specification. An instruction has a given type, defined by a `format` definition, which describes the binary representation of the instruction [Per25].

The format is represented as a bitfield with named and typed member fields. Additionally, a format can hold

- *field access functions*, which allow access to specific format fields applied to a specified function. The field access function serves as a separation between the instruction's behavior and its decoding. [Per25]
- *encoding functions*, which describe how a given immediate has to be encoded into the binary representation. These are generally automatically generated by the LCB, if possible, as they represent the inverse of the *field access function*. [Per25]
- *predicate functions*, which verify that a given immediate is valid. These are generally automatically generated by the LCB, if possible. [Per25]

An instruction is defined by the `instruction` definition and describes the behavior of the instruction, potentially accessing format fields [FHH+25a].

Listing 2.8 shows an example of a ISA specification with a single `format` definition and two instruction definitions, `ADDXI` and `ADDWI`, both using the defined format. The format further specifies a field access function and the optional encoding and predicate functions.

```
1  instruction set architecture AArch64 = {
2
3    format AddSubImmFormat: Instr =
4    {
5      sf       [31],
6      ff       [29],
7      op       [30, 28..23],
8      sh       [22],
9      imm12          [21..10] : Bits<12>,
10     rn       [9..5] : Bits<5>,
11     rd       [4..0] : Bits<5>,
12     imm12X = imm12 as Bits<64>, // field access function / decoding
        function
13     imm := imm12X(11..0), // encoding function
14     imm12X :- imm12X >= -2048 && imm12X <= 2047 // predicate function
15   }
16
17   instruction ADDXI : AddSubImmFormat =
18     let result, flags = VADL::adds(X(rn), imm12X as Bits<64>) in
19       { X(rd) := result }
20
21   instruction ADDWI : AddSubImmFormat =
22     let result, flags = VADL::adds(W(rn), imm12X as Bits<32>) in
23       { W(rd) := result }
24
25 }
```

Listing 2.8: Example VADL ISA specification, defining an instruction format and two associated instructions.

2.2.3 Registers

Register files and single registers are defined in the ISA specification [FHH+25a]. Listing 2.9 shows the definition of a single register file and 3 aliases. Line 1 depicts a mapping `S` of a 5-bit register index to the register type `Bits<64>`, which is a bit vector type with a bit width of 64.

Line 4 shows an alias definition `X`, which is additionally annotated, declaring the register `X31` as a zero register. Zero registers ignore writes and always return 0 on reads.

Line 7 declares an alias register file, which represents a 32-bit view into the 64-bit registers defined by `X`. In addition to the zero-register annotation, the definition is annotated with `[ overwrite source : zero ]`, controlling how the upper bits of the underlying register, `S`, should be updated on writes to the 32-bit view. The default behavior is a partial write, which preserves the upper bits of the super-register, while `zero` indicates that the upper bits should be zeroed and `sign` indicates that the written value is sign-extended to the full width of the super-register.

Finally, line 10 declares a single register alias, which references register 31 of the register file `S`. While the alias definition of `X` declares the 32nd register of `S` as a zero-register, `SP` does not specify any additional behavior on the alias. Therefore, reads and writes to SP represent reads and writes to the 32nd register of `S` with no additional behavior, as if `S` were directly written to or read from.

```
1  register        S: Bits<5> -> Bits<64>
2
3  [ zero : X(31) ]
4  alias register  X = S
5
6  [ zero : W(31) ]
7  [ overwrite source : zero ]
8  alias register  W = X(*)(31..0)
9
10 alias register  SP : Bits<64> = S(31)
```

Listing 2.9: Example VADL specification, defining 3 register files and an individual register.

2.3 AArch64 Architecture

AArch64, also known as ARM64, is a 64-bit RISC architecture. AArch64 provides 31 general purpose registers, where `X0` to `X30` represent the 64-bit views of these registers and `W0` to `W30` the 32-bit views. Additionally, the 64-bit register `XZR` and its 32-bit equivalent `WZR` always read 0 and ignore writes [Lim25].

Additionally, AArch64 defines a collection of flags in the so-called `PSTATE` (processor state), which holds information about the most recent ALU operations and other processor-relevant information [Lim].

AArch64 defines the condition flags listed in table 2.2, abbreviated `NZCV`, which are updated in the `PSTATE` after every ALU instruction [Lim26e]:

Abbreviation	**Name**	**Description**
N	Negative	Set to 1 if the result of the last flag-setting instruction was negative.
Z	Zero	Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
C	Carry	Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.
V	Overflow	Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example, a signed overflow on an addition.

Table 2.2: AArch64 condition flags.

These conditions are further used for conditional instructions, including conditional branching (`b.eq`, `b.ne`, and similar) or conditional select (`csel`) [Lim26a] [Lim26b].

OpenVADL, introduced in section 2.2, includes a specification for a subset of the AArch64 architecture [Opea], which is called *AArch64 specification* throughout this thesis. Additionally, OpenVADL specifies a reduced variant of the AArch64 specification, the *Xaarch64 specification* [Opeb], which is a 64-bit-only version of the *AArch64 specification*.

2.4 Related Work

This thesis builds on the work of Per [Per25], who implemented the original LCB for OpenVADL. His thesis also introduces the concept of a semantic gap between a VADL specification and the structure LLVM requires a target to be described in, which this thesis adopts to describe why the LCB has to derive and interpolate information not explicitly present in a specification.

Gisy [Gis] also worked on the LCB, similarly to this thesis. His work focuses on performance problems that surfaced while working with the RISC-V specification, whereas this thesis works in parallel, focusing on problems that surfaced while working with the AArch64 specification.

CHAPTER 3

Implementation

This chapter introduces problems that surfaced while working on the AArch64 and Xaarch64 specifications, and their respective causes and improvements implemented in the LCB to overcome these defects. Each section focuses on a specific topic of problems and their resolution.

3.1 Register and Subregister Hierarchy

3.1.1 Subregister Hierarchy Generation

As stated in section 2.1, LLVM requires an explicit definition of a target's register files, including how narrower and wider views of the same register file relate to each other as subregisters [Prok]. Listing 3.1 shows an example of how registers and register files and their relationships are defined in OpenVADL, taken and adapted from the AArch64 specification, using a base register file `S`, its full-width alias `X`, and the two narrower views `R` and `W`.

```
1 using           Index = Bits<5>
2
3 register        S : Index -> Bits<64>
4
5 alias register  X = S
6 alias register  R = S(*)(31..0)
7 alias register  W = X(*)(31..0)
```

Listing 3.1: Example register and subregister file definitions in OpenVADL

A definition such as `S` in Listing 3.1 declares an entire register file, indexed by `Index`, rather than a single register. In the generated LLVM backend, this register file is exploded

into one separate physical register definition per index, and the relationships between the registers declared in the VADL specification, such as X being an alias of S, or W being derived from X, are represented as a register-subregister hierarchy, generated once for every index. Before this work, the generator produced the hierarchy shown in Figure 3.1, where X, W, and R were all generated as direct subregisters of S.

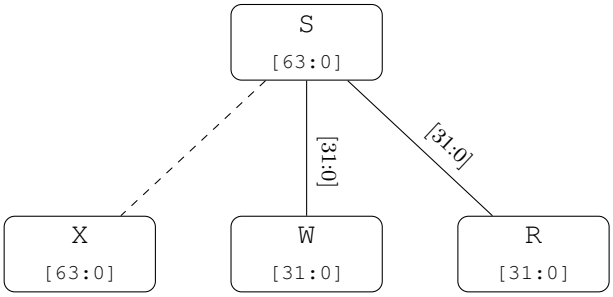

Figure 3.1: Register-subregister hierarchy before this work. X, W, and R are all direct subregisters of S.

This causes problems when one subregister class needs to access another subregister class of the same base register class. Concretely, a value held in the X register class could not be extracted or truncated into the W register class, even though the VADL specification explicitly derives W from X (Listing 3.1). LLVM's EXTRACT_SUBREG operation requires the target register class to be reachable from the source register class by descending through the hierarchy along subregister indices, as shown in Figure 3.1. Since W was only reachable from S, not from X, no such index existed between X and W, and the truncation could not be generated.

Because of this, the generation of the register-subregister hierarchy was reworked so that every register is linked to the next-larger, or identically sized, register that contains its bits, rather than every alias being attached directly to a single common base register. Figure 3.2 shows the resulting hierarchy generated for listing 3.1. X remains a same-width subregister of S, but W is now a subregister of X rather than of S directly. R, although declared from S in the VADL specification, is generated as a same-width subregister of W, since R and W occupy exactly the same bits. This gives every pair of overlapping registers a subregister-index path between them, direct or composed across multiple steps, and allows EXTRACT_SUBREG and INSERT_SUBREG to be generated between any two related register classes.

3.1.2 Truncation and Extension Instruction-Selection Patterns

To make use of the adapted hierarchy described in section 3.1.1, truncation and extension patterns are generated for every pair of registers connected by a subregister relationship. This allows LLVM's instruction selection to truncate registers to any subregister in the register hierarchy, or extend a subregister to any parent register in its register hierarchy.

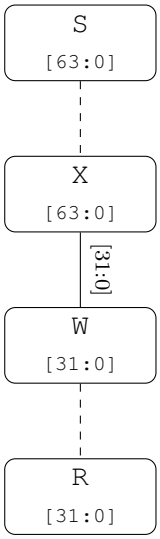

Figure 3.2: Register-subregister hierarchy after this work. `W` is a direct subregister of `X`, while `R` remains a same-width subregister of `W`.

Listing 3.2 shows the generated truncation patterns for listing 3.1, which use `EXTRACT_SUBREG` to obtain the lower 32 bits of an `S` or `X` register as a 32-bit value. Listing 3.3 shows the corresponding extension patterns, which use `INSERT_SUBREG` to place an `R` or `W` value into the corresponding super-register class.

```
1 def : Pat<
2   (i32 (trunc S:$rd)),
3   (EXTRACT_SUBREG S:$rd, SUB_32)>;
4
5 def : Pat<
6   (i32 (trunc X:$rd)),
7   (EXTRACT_SUBREG X:$rd, SUB_32)>;
```

Listing 3.2: Example of truncation patterns generated for the AArch64 specification, allowing access to subregisters of a given register.

```
1 def : Pat<
2   (i64 (anyext R:$rd)),
3   (INSERT_SUBREG (i64 (IMPLICIT_DEF)), R:$rd, SUB_32)>;
4
5 def : Pat<
6   (i64 (anyext W:$rd)),
7   (INSERT_SUBREG (i64 (IMPLICIT_DEF)), W:$rd, SUB_32)>;
```

Listing 3.3: Example of extension patterns generated for the AArch64 specification, allowing access to super-registers of a given subregister.

3.1.3 Memory Instruction-Selection Patterns

In addition to the truncation and extension patterns described in the previous section, additional patterns are generated for LLVM's `store` and `truncstore` SelectionDAG nodes, which describe copying a register's value to memory and possibly truncating the value.

AArch64's `STRB` instruction, for example, is defined in the OpenVADL specification using the `W` register class. Listing 3.4 shows the patterns generated directly from such a definition, allowing LLVM's instruction selector to copy a value held in a `W` register to memory, matching how `STRB` is defined in the AArch64 specification [Lim26c].

```
1 def : Pat<(truncstorei8 W:$rt, S:$rn),
2         (STRB S:$rn, W:$rt, (i64 0))>;
3
4 def : Pat<(store W:$rt, AddrFI:$rn),
5         (STURW AddrFI:$rn, W:$rt, (i64 0))>;
```

Listing 3.4: Example of an 8-bit truncating and non-truncating store patterns for the AArch64 specification.

Generating patterns only for the declared register class would require the specification author to additionally define a separate variant for every super-register a value might actually be held in, which defeats the purpose of deriving this behavior automatically from the specification. Instead, the patterns from Listing 3.4 are automatically extended to `W`'s super-register, `S`, as shown in Listing 3.5, without requiring any additional information from the specification.

```
1 def : Pat<(truncstorei8 S:$rt, S:$rn),
2         (STRB S:$rn, (EXTRACT_SUBREG S:$rt, SUB_32), (i64 0))>;
3
4 def : Pat<(truncstorei32 S:$rt, AddrFI:$rn),
5         (STURW AddrFI:$rn, (EXTRACT_SUBREG S:$rt, SUB_32), (i64 0))>;
```

Listing 3.5: Truncating and non-truncating store patterns extended to `W`'s super-register `S`.

These additional patterns allow a value held in `S`, the parent register of `W`, to be stored to memory using the same `STRB` and `STURW` instructions, even if only the `W` case is explicitly declared in the OpenVADL specification.

3.2 Immediate Type Mismatch between VADL and LLVM

As stated in section 2.2.2, VADL format fields can have arbitrary bit widths, while TableGen and the C++ code it generates only support power-of-two integer types starting at 8 bits. The generator has to choose an LLVM type for every immediate

present in a specification, and this type must be considered legal by LLVM's SelectionDAG type inference, as otherwise corresponding patterns using these immediates cannot be selected or prevent a successful compilation.

Currently, this discrepancy between the type of an immediate as stated in an OpenVADL specification and the generated LLVM type by the LCB either causes performance problems or, in the case of the `Xaarch64` specification, a build failure of the generated compiler.

Gisy [Gis] independently encountered the same discrepancy and addressed it with a temporarily implemented annotation in the RV64IM OpenVADL specification and left a general solution to future work, which is addressed here. The annotation is not ideal, as stated by Gisy [Gis], as it forces implementation details of the LCB and LLVM onto the specification author.

Listing 3.6 shows an example of an immediate generated for the `MOVZXPos00` instruction, a 16-bit move-with-zero-extend instruction in the Xaarch64 specification.

```
1 class AArch64Base_MOVZXPos00_imm16<ValueType ty> : Operand<ty>
2 {
3   let EncoderMethod = "AArch64Base_MOVZXPos00_wrapper";
4   let DecoderMethod = "AArch64Base_MOVZXPos00_imm16_decode_wrapper";
5 }
6
7 def AArch64Base_MOVZXPos00_imm16AsInt16
8     : AArch64Base_MOVZXPos00_imm16<i16>
9     , ImmLeaf<i16, [{ return AArch64Base_MovFormat_imm16_predicate(
    Imm); }]>;
```

Listing 3.6: Example of a generated immediate for the Xaarch64 specification before this work, using a 16-bit immediate type

The immediate's LLVM type is derived from the width of the underlying field, 16 bits, and emitted as `i16`. The `Xaarch64` specification has no register narrower than 64 bits, so `i16` was never registered as a legal type for this target. TableGen's type inference for `zext` requires the source type to be among the target's legal types [Proi]. This is not the case for the `Xaarch64` specification, which results in an error while compiling the generated compiler.

To fix this, immediates are no longer typed by the width of their underlying field, but by the smallest register width actually present in the architecture. In the case of the `Xaarch64` specification, this is 64 bits. Listing 3.7 shows the same immediate with the changes implemented in this work.

```
1 def AArch64Base_MOVZXPos00_imm16AsInt64
2     : AArch64Base_MOVZXPos00_imm16<i64>
```

```
3     , ImmLeaf<i64, [{ return AArch64Base_MovFormat_imm16_predicate(
      Imm); }]>;
```

Listing 3.7: Example of a generated immediate after this work, using the architecture's minimum register width as the immediate type

The immediate still encodes the same 16-bit value, but its LLVM type changed from `i16` to `i64`, which is a legal type for the `Xaarch64` specification. This resolves the TableGen type inference failure without requiring any change to the VADL specification. With this heuristic, the annotation mechanism introduced by Gisy as a temporary workaround is no longer needed.

3.3 Predicate Function Generation

As described in section 2.2.2, every immediate of a given instruction is validated by an associated predicate function in the generated compiler, which is deduced from the instruction's definition in an OpenVADL specification.

LLVM's generated instruction selector encodes its matching logic as a compact bytecode table [Proc]. Predicate checks within this table reference their corresponding predicate function through a single byte index into a generated array of function pointers. Since a single byte can only address 256 entries, the generated matcher cannot reference more than 256 predicate functions. The AArch64 specification currently generates 505 predicate functions, which exceeds this limit.

Since a single instruction format is typically shared by many instructions, many of them require the same predicate function. Currently, the LCB generates a predicate function separately for every instruction, regardless of whether an identical one has already been generated for another instruction sharing the same format and field access. As a result, the number of generated predicate functions scales with the number of instructions referencing an immediate, not with the number of actually distinct functions needed. Because of these duplicated generated predicate functions, the compiler generated by the AArch64 specification fails to compile.

Listing 3.8 shows an excerpt of two predicate functions generated for the instructions `ADDXLSL` and `ADDXLSR` of the AArch64 specification. Both instructions use the same format in the specification, but generate two identical functions.

```
1 static bool AArch64Base_ADDXLSL_imm6_predicate(uint64_t imm6) {
2    return (
3     VADL_and(
4       VADL_sgeq(imm6, 6, ((int64_t) 0xfffffffffffffe0 ), 64),
5       1,
6       VADL_sleq(imm6, 6, ((int64_t) 0x1f ), 64),
7       1)
8       ? ((bool) 0x1 )
```

```
9       : ((bool) 0x0 ));
10  }
11  static bool AArch64Base_ADDXLSR_imm6_predicate(uint64_t imm6) {
12     return (
13      VADL_and(
14        VADL_sgeq(imm6, 6, ((int64_t) 0xfffffffffffffe0 ), 64),
15        1,
16        VADL_sleq(imm6, 6, ((int64_t) 0x1f ), 64),
17        1)
18        ? ((bool) 0x1 )
19        : ((bool) 0x0 ));
20  }
```

Listing 3.8: Excerpt of two identical generated predicate functions of the AArch64 specification.

To circumvent this, predicate functions are no longer generated per instruction, but deduplicated per format and field access. Since two immediates sharing the same format and field access always produce an identical predicate function, only one such function is generated and reused by every instruction referencing it, instead of generating one per referencing instruction. This reduces the number of generated predicate functions for the AArch64 specification from 505 to 31, allowing the generated compiler to be compilable again.

3.4 Machine-Instruction Label Classification

As discussed in section 2.2.1, there is a semantic gap between the specification of an architecture in OpenVADL and LLVM. Information present in the VADL specification needs to be interpolated to fit LLVM's target model.

One such case is the procedure call. All procedure calls share the same defining behavior: control is transferred to a target address by writing to the program counter, and the address to resume execution at afterwards is preserved so that it can be returned to later [HP94a]. The return address may be stored in a dedicated link register, as on AArch64 [Lim26d] and RISC-V [AW], or in a location on the call stack, as on x86 [Cor24]. Architectures additionally differ in how the target address itself is computed, e.g., as a plain register value, or as the sum of a register and an immediate offset. LLVM needs this behavior made explicit for a given target before it can translate an indirect function call present in some LLVM-IR into that target's instructions.

Another case concerns conditional-select operations. LLVM's SelectionDAG represents these with an explicit condition code attached to a target-independent select node [Proe]. A target implementation has to provide the actual instructions to which these condition codes are translated.

Both cases raise the same need: instructions defined in OpenVADL have to be classified and labelled, so that the LCB can handle them separately and emit the patterns LLVM

requires.

3.4.1 Indirect-Call Lowering

Because the structure of a call instruction (where the return address goes, how the target address is computed) varies across architectures, a single generic label cannot capture every architecture's call instruction. Instead, the LCB defines one label per structurally distinct variant. `JALR` and `JAL` cover RISC-V-style calls, where the target address is computed from a register plus an immediate (`JALR`) or from the program counter plus an immediate (`JAL`), and the return address is written to a register file. `J` and `JR` are the corresponding immediate- and register-based jumps without linking.

AArch64's `BLR` instruction [Lim26d] extends this set. It is structurally identical to the RISC-V-style `JR` in that the program counter is written directly from a register read, with no immediate involved. However, unlike `JR`, it additionally writes the return address to a register file.

`BLR` falls into a gap between the labels `JR` and `JALR`. It shares `JR`'s addressing mode (a register read, no immediate) but, like `JALR`, also writes a return address. Because no existing label covered a register-only call that also links, `BLR` matched none of them before this work.

Labelling instructions whose behavior structurally matches `BLR` allows the LCB to emit branch patterns for these architectures, which in turn enables LLVM to translate indirect function calls into target-dependent instructions.

3.4.2 Conditional Select Operations

The condition of a conditional select instruction is expressed as a boolean formula over one or more of the architecture's status flags. Either a single flag is tested directly (e.g. `Carry == 1`), or two flags are compared against each other (e.g. `Negative == Overflow`). OpenVADL marks the registers such formulas may use with special annotations [FHH+25a], e.g. `[carry status register]`, which allows the generator to determine which status flags a given instruction's condition depends on. The annotations provide the necessary information needed to deduce conditions based on the registers used.

Table 3.1 lists the select-operation labels introduced as part of this work, together with the status-flag formula each one captures.

As Table 3.1 shows, the labels for signed conditions mirror standard mathematical relational operators directly, e.g., `SGTH` for signed greater-than. This is possible because the `Negative` and `Overflow` flags are defined the same way on every architecture that implements flag-based comparison [HP94b].

The same is not true for unsigned conditions, because whether the carry flag is set on subtraction is a convention each architecture chooses independently. x86 sets the

Label	Condition	Status-flag formula
EQ	Equal	Zero = 1
NEQ	Not Equal	Zero = 0
SLTH	Signed Less-Than	Negative ≠ Overflow
SGEQ	Signed Greater-or-Equal	Negative = Overflow
SGTH	Signed Greater-Than	Negative = Overflow **and** Zero = 0
SLEQ	Signed Less-or-Equal	Negative ≠ Overflow **or** Zero = 1
CS	Carry Set	Carry = 1
CC	Carry Clear	Carry = 0
NS	Negative Set	Negative = 1
NC	Negative Clear	Negative = 0
VS	Overflow Set	Overflow = 1
VC	Overflow Clear	Overflow = 0

Table 3.1: Status-flag formulas for the conditional-select classification labels (CSEL_<Label>). Ten of the twelve labels, all except EQ and NEQ, were introduced as part of this work. Each label exists as a 32-bit and a 64-bit register-width variant.

Carry flag on a borrow [Cor24]. The operator unsigned-less-than would be implemented by comparing the Carry flag with one. AArch64, in contrast, sets the Carry flag on the absence of a borrow [Lim26e], so the same flag value instead expresses unsigned greater-or-equal. A label such as ULTH (unsigned-less-than) would therefore assert a comparison result that only holds under one of the two conventions.

For this reason, unsigned comparisons are not expressed as labels but as the flag tests themselves (e.g. carry-set (CS) or carry-clear (CC)).

Correctly identifying an architecture's conditional select instructions allows the LCB to generate the necessary patterns for LLVM to translate from its own LLVM-IR to target-dependent instructions, as can be seen in listing 3.9, which shows the generated patterns for the AArch64 specification.

```
1  switch(CC) {
2    case ISD::CondCode::SETEQ:
3      return SDValue(DAG.getMachineNode(aarch64temp::CSELEQX, dl, MVT::
       i64, TVal, FVal, ConditionFlag ), 0);
4
5    case ISD::CondCode::SETNE:
6      return SDValue(DAG.getMachineNode(aarch64temp::CSELNEX, dl, MVT::
       i64, TVal, FVal, ConditionFlag ), 0);
7
8    case ISD::CondCode::SETLT:
9      return SDValue(DAG.getMachineNode(aarch64temp::CSELLTX, dl, MVT::
       i64, TVal, FVal, ConditionFlag ), 0);
10
11   case ISD::CondCode::SETGT:
```

```
12      return SDValue(DAG.getMachineNode(aarch64temp::CSELGTX, dl, MVT::
        i64, TVal, FVal, ConditionFlag ), 0);
13
14    case ISD::CondCode::SETGE:
15      return SDValue(DAG.getMachineNode(aarch64temp::CSELGEX, dl, MVT::
        i64, TVal, FVal, ConditionFlag ), 0);
16
17    case ISD::CondCode::SETLE:
18      return SDValue(DAG.getMachineNode(aarch64temp::CSELLEX, dl, MVT::
        i64, TVal, FVal, ConditionFlag ), 0);
19    default:
20      llvm_unreachable("unimplemented operand");
21  }
```

Listing 3.9: Example of the generated C++ code to map from LLVM's target-independent to AArch64's conditional select instructions.

CHAPTER 4

Evaluation

This chapter showcases the evaluation of the changes and implementations from section 3. Section 4.1 introduces the setup used to evaluate the impact of the changes done in this work. Sections 4.2, 4.3, 4.4, and 4.5 show the impact of the respective sections in 3, while 4.6 demonstrates the overall impact of all changes.

4.1 Setup

The AArch64 and Xaarch64 specifications were used as a basis for all evaluations and implementations, as mentioned in section 3. Furthermore, other specifications present in the OpenVADL repository were used to verify that no regressions were introduced.

Implementations and changes to the LCB were tested and evaluated with the following setup: OpenVADL was used to generate an LLVM compiler artefact, the source code of the generated compiler. This generated compiler was then compiled using its own generated build files. For problems that prevented the generated compiler from being built at all, successfully compiling it constituted the entire evaluation.

If the generated compiler was successfully compiled, a set of C source programs, representing the test suite, was used for further evaluation of the compiler's capabilities. An attempt was made to compile each individual C program without specific optimizations enabled. If the compilation succeeded, the resulting executable was executed inside a Linux virtual machine based on the AArch64 architecture to evaluate the correctness of the compiled program.

Table 4.1 lists the test files used, along with information on whether the specific test file makes use of the C standard library `libc`, and a description of the capability it is intended to test [Opec] [Oped]. Most of the test files are kept simple and focus on one specific capability the generated compiler should support. The test files were already part of the OpenVADL repository before this work and were not created in this work.

Filename	Uses Libc	Description
add.c	No	Add two constants and verify the result.
sub.c	No	Subtract two constants and verify the result.
mul.c	No	Multiply two constants and verify the result.
mulu64.c	No	Perform multiple multiplications and additions with large unsigned-64-bit integers and verify the result.
geq.c	No	Compare two constants with the *greater-than-or-equal* operator.
gt.c	No	Compare two constants with the *greater-than* operator.
leq.c	No	Compare two constants with the *less-than-or-equal* operator.
lt.c	No	Compare two constants with the *less-than* operator.
global.c	No	Index a globally defined array and verify the indexed element.
indirect_call_min.c	No	Call an identity function referenced through a pointer.
indirect_call.c	No	Call an comparison function referenced through a pointer.
large_constant_with_compare.c	No	Add a large constant to a given variable and verify the result.
large_constant.c	No	Compare two large constants with each other.
rem.c	No	Use the modulo operator to compute the remainder of a division and verify the result.
simple.c	No	Return a constant from the main function.
switch.c	No	Define an array of function pointers (*jump table*) and call the appropiate functions.
binaryFirst.c	Yes	Binary search for the first (leftmost) index of a target value in a sorted array, using a comparator function pointer, and verify the result.
binaryLast.c	Yes	Binary search for the last (rightmost) index of a target value in a sorted array, using a comparator function pointer, and verify the result.
insertionSort.c	Yes	Implementation of insertion sort, additionally allowing partial sorts.
rotate.c	Yes	Rotate the values in an array.

Table 4.1: Table of test files used for the implementation and evaluation.

4.2 Register and Subregister Hierarchy

Reworking the register-subregister hierarchy as described in section 3.1 affected the most test files of all implemented topics and especially allowed more complex test files, like binary search or insertion sort, to successfully compile and execute correctly.

As described in the respective implementation section, before this work, LLVM could not correctly extend or truncate a register to its super- or subregister. The exact compile error LLVM panicked with is shown in listing 4.1.

```
1 Assertion failed: (RC && "No legal register class for VT supports
     that SubIdx"), function ConstrainForSubReg, file InstrEmitter.cpp,
      line 493.
```

Listing 4.1: Excerpt of the error LLVM panicked with, while compiling the test file *indirect_call.c*, before this work.

As partially stated in the error in listing 4.1, LLVM was unable to select a register of register class `W` given a register of register class `X`, as no subregister-index was defined for this non-existent register class relationship.

Table 4.2 shows the results of compiling and executing a subset of the mentioned test files, namely test files that were affected by the change.

Filename	Before		After	
	Compilable	Correct	Compilable	Correct
global.c	No	-	Yes	Yes
indirect_call_min.c	No	-	Yes	Yes
indirect_call.c	No	-	Yes	Yes
large_constant_with_compare.c	No	-	Yes	Yes
large_constant.c	No	-	Yes	Yes
switch.c	No	-	Yes	Yes
binaryFirst.c	No	-	Yes	Yes
binaryLast.c	No	-	Yes	Yes
insertionSort.c	No	-	Yes	Yes
rotate.c	No	-	Yes	No

Table 4.2: Table of affected files by the introduced changes, comparing the compilation and execution status before and after this work.

4.3 Immediate Type Mismatch between VADL and LLVM

As stated in the accompanying implementation section 3.2, the immediate type mismatch between OpenVADL and LLVM caused the generated compiler for the Xaarch64 speci-

fication to be uncompilable. Listing 4.2 shows an excerpt of the error LLVM panicked with.

```
1 Type set is empty for each HW mode:
2 possible type contradiction in the pattern below (use -print-records
     with llvm-tblgen to see all expanded records).
3 vtInt:  (vt:{ *:[Other] })
4 Generated from record:
5 vtInt { // SDPatternOperator PatFrags PatFrag PatLeaf
6   list<SDNodeProperty> Properties = [];
7   dag Operands = (ops);
8   ...
```

Listing 4.2: Excerpt of the compilation error for the generated compiler of the Xaarch64 specification, before this work.

As stated in the reported error in listing 4.2, LLVM ran into a contradiction because the intersection set between the set of legal types and the set of types defined for the specific immediate responsible for the error was empty. The set of legal types was {i64}, while the set of types of the immediate was {i16}.

As the error prevented the compiler from compiling, no changes to specific test files are shown. With the changes to immediate typing introduced in this work, the generated compiler successfully compiles.

4.4 Predicate Function Generation

As stated in the accompanying implementation section 3.3, the number of generated predicate functions prevented the generated compiler for the AArch64 specification from compiling. Listing 4.3 shows an excerpt of the error that LLVM panicked with.

```
1 openvadl/output/lcb/llvm-final/build/lib/Target/aarch64temp/
     aarch64tempGenDAGISel.inc:7787:34: error: constant expression
     evaluates to 256 which cannot be narrowed to type 'unsigned char'
     [-Wc++11-narrowing]
2  7787 | /* 12105*/   OPC_CheckPredicate, 256, //
     Predicate_AArch64Base_CCMNICCX_immXAsInt64
3       |                                  ^~~
```

Listing 4.3: Excerpt of the compilation error for the generated compiler of the AArch64 specification, before this work.

The reported error in listing 4.3 shows an excerpt of LLVM's bytecode table used for instruction selection. As more than 256 predicate functions were defined before this work, LLVM could no longer store an index to the excess functions, and compilation failed.

This work reduced the number of generated predicate functions from 505 to 31.

Similar to section 4.3, no changes to specific test files are shown, as the error prevented the generated compiler from compiling. With the reduction of the number of generated predicate functions introduced in this work, the generated compiler successfully compiles.

4.5 Machine-Instruction Label Classification

4.5.1 Indirect-Call Lowering

Introducing the necessary labeling and classification logic for instructions resembling `BLR`'s behavior allowed the necessary indirect call instruction patterns to be generated, as described in 3.4. This in turn allowed test files making use of function pointers and indirect calls to successfully compile and execute correctly.

As described in the respective implementation section, before this work, LLVM could not match a SelectionDAG node representing a procedure call to an address stored in a register to AArch64's native `BLR` instruction. This can be seen in the listing 4.4, which depicts the error LLVM panicked with when trying to compile the test file *indirect_call.c*.

```
1 fatal error: error in backend: Cannot select: t19: ch,glue =
     aarch64tempISD::CALL t17, t10, RegisterMask:Untyped, indirect_call.
     c:17:10
2   t10: i64,ch = load<(dereferenceable load (s64) from %ir.f_sub)> t9,
      FrameIndex:i64<1>, undef:i64, indirect_call.c:17:10
3     t7: i64 = FrameIndex<1>
4     t2: i64 = undef
5   t18: Untyped = RegisterMask
6 In function: driver
```

Listing 4.4: Excerpt of the error LLVM panicked with, while compiling the test file *indirect_call.c*, before this work.

Table 4.3 shows the results of compiling and executing a subset of the mentioned test files, namely test files that were affected by the change.

As can be seen in table 4.3, the listed test files are still uncompilable after the introduced changes. This is caused by the missing implementations of the register-subregister hierarchy adaptation, as described in 3.1, and the introduction of conditional select labels, as described in 3.4.

Together with the mentioned changes, these test files are compilable and are partially correctly executed, as can be seen in section 4.6.

4.5.2 Conditional select operations

Extending the necessary labeling and classifying logic for instructions implementing conditional select operations allowed the necessary mapping between LLVM's target-independent and target-dependent conditions to be generated, as described in 3.4. This

Filename	Before		After		Notes
	Compilable	Correct	Compilable	Correct	
indirect_call_min.c	No	-	No	-	additionally requires the changes introduced in 3.1
indirect_call.c	No	-	No	-	additionally requires the changes introduced in 3.1
switch.c	No	-	No	-	additionally requires the changes introduced in 3.1
binaryFirst.c	No	-	No	-	additionally requires the changes introduced in 3.4
binaryLast.c	No	-	No	-	additionally requires the changes introduced in 3.4
insertionSort.c	No	-	No	-	additionally requires the changes introduced in 3.4

Table 4.3: Table of affected files by the introduced changes, comparing the compilation and execution status before and after this work.

resulted in test files making use of conditional select operations with a variety of conditions becoming compilable.

As described in the respective implementation section, before this work, LLVM could not match various conditions, including *signed greater-than* or *signed less-than*, to their target-dependent counterpart, resulting in the compilation failure of test files. LLVM panicked with the error message `unimplemented operand`.

Table 4.4 shows the results of compiling and executing a subset of the mentioned test files, namely test files that were affected by the change.

Similar to the results of section 4.5.1, the compilation status of the affected test files did not change, as the changes to the register-subregister hierarchy, introduced in 3.1, are additionally needed.

Filename	Before		After		Notes
	Compilable	Correct	Compilable	Correct	
binaryFirst.c	No	-	No	-	additionally requires the changes introduced in 3.1
binaryLast.c	No	-	No	-	additionally requires the changes introduced in 3.1
insertionSort.c	No	-	No	-	additionally requires the changes introduced in 3.1
rotate.c	No	-	No	-	additionally requires the changes introduced in 3.1

Table 4.4: Table of affected files by the introduced changes, comparing the compilation and execution status before and after this work.

Together with the mentioned changes, these test files are compilable and are partially correctly executed, as can be seen in section 4.6.

4.6 Overall Results

As most test files required multiple changes to the implementation to be compilable and correctly executable, table 4.5 shows all test files, as introduced in 4.1, and their before-and-after compilation, as well as execution status.

As can be seen, the first 8 test files were already compilable and were not affected, nor targeted, by the changes made in this work. Additionally, `mulu64.c` and `rem.c` are still uncompilable after the implemented changes. The necessary changes to the implementation are out of the scope of this work and mentioned in 5.1.

The remaining 10 test files successfully compile and execute now, except for `rotate.c`, which, too, is out of scope for this work.

While `rotate.c` compiles correctly, it crashes with a segmentation fault at runtime. This is caused by the LCB incorrectly handling certain negative 64-bit integer constants, which results in a corrupted value in the generated assembly, triggering the crash. This is an open bug in the LCB.

Filename	Before		After	
	Compilable	Correct	Compilable	Correct
add.c	Yes	Yes	Yes	Yes
sub.c	Yes	Yes	Yes	Yes
mul.c	Yes	Yes	Yes	Yes
geq.c	Yes	Yes	Yes	Yes
gt.c	Yes	Yes	Yes	Yes
leq.c	Yes	Yes	Yes	Yes
lt.c	Yes	Yes	Yes	Yes
simple.c	Yes	Yes	Yes	Yes
mulu64.c	No	-	No	-
rem.c	No	-	No	-
global.c	No	-	Yes	Yes
indirect_call_min.c	No	-	Yes	Yes
indirect_call.c	No	-	Yes	Yes
large_constant_with_compare.c	No	-	Yes	Yes
large_constant.c	No	-	Yes	Yes
switch.c	No	-	Yes	Yes
binaryFirst.c	No	-	Yes	Yes
binaryLast.c	No	-	Yes	Yes
insertionSort.c	No	-	Yes	Yes
rotate.c	No	-	Yes	No

Table 4.5: Compilation and execution status of every test file, before and after this work.

CHAPTER 5

Future Work

5.1 Missing Extension Patterns

5.1.1 Register Extension Patterns

As can be seen in section 4.6, after the changes made in this work, there are still two test files that do not compile. One reason is missing *zero-extension* patterns for subregisters to their respective super-registers. Section 3.1 introduced *any-extension* patterns (see 2.1 for an explanation) for register-subregister hierarchies. Similar patterns are needed for *zero-* and *sign-extension* operations.

In contrast to *any-extension*, *zero-* and *sign-extension* cannot simply be mapped to `INSERT_SUBREG` operations. Any-extension leaves the upper bits of the extended register unconstrained, so it is sufficient to tell LLVM to treat the subregister as its superregister, regardless of what value the upper bits hold. Zero- and sign-extension do constrain these upper bits, requiring them to be zero-filled or to repeat the sign bit. Satisfying this constraint requires an instruction that actually sets the upper bits, rather than simply reinterpreting the subregister as its superregister. Since these instructions are target-dependent, these patterns cannot be generated without taking the architecture's available instructions into account.

The LCB has the ability to recognize necessary instructions with the help of machine instruction label classification introduced in 2.2. Additionally, OpenVADL provides the `[overwrite source: ]` annotation describing the effect of writes to subregisters on their super-registers.

5.1.2 Memory Load Patterns

Section 3.1.3 introduced additional patterns for `store` and `truncstore` instructions, which were deduced from existing store patterns and extended to available superregisters

of the registers used in the patterns. A truncating-store pattern for an `i8` stored in a `W` register additionally generated a pattern for an `i8` stored in an `X` register, `W`'s superregister.

Similar additional patterns should be generated for load instructions, including load-with-extension instructions such as `zextload`, `extload` and `sextload`.

Given a pattern that maps the instruction `zextloadi8`, loading and zero-extending a byte to 32 bits, additional patterns for bit-sizes with the power of 2 should be generated. Namely `zextloadi8` to 64-bits. Analogous patterns should be generated for `sextload` and `extload`.

Listing 5.1 demonstrates this, with the first pattern being generated directly from the OpenVADL specification, as implicitly written by the specification author, while the next pattern is deduced by the LCB.

```
1 def : Pat<(i32 (zextloadi8 S:$rn)),
2         (LDRB S:$rn, (i64 0))>;
3
4 def : Pat<(i64 (zextloadi8 S:$rn)),
5         (SUBREG_TO_REG (i64 0), (LDRB S:$rn, (i64 0)), SUB_32)>;
```

Listing 5.1: Generated patterns for the `zextloadi8` LLVM-IR instruction. The first pattern is currently generated by the LCB, while the second pattern is missing.

5.2 Machine-Instruction Labels

Section 2.2.1 introduced the concept of the *semantic gap*, while section 3.4 extended the available machine instruction labels.

5.2.1 Multiple Labels

A current limitation of the machine instruction label classification is that a single instruction can only be classified with a single label. This restricts the LCB from taking advantage of instructions that cover the behavior of multiple labels.

Allowing multiple labels per instruction would increase the flexibility of the LCB in generating necessary LLVM patterns.

5.2.2 Unsigned Comparisons

As described in section 3.4, labels for unsigned comparisons are not available. This is not limited to conditional select instructions, but also to conditional branch instructions.

Because of the missing labels, necessary mappings between LLVM instructions using unsigned comparisons (like unsigned-less-than, unsigned-less-or-equal) and associated target-dependent instructions are not generated.

5.3 Optimizations

As described in section 4, the C test files were compiled with no optimizations enabled.

LLVM provides several target-independent optimization passes, including tail duplication, branch folding, and block placement [Prof] [Proh] [Prog]. To reason about a function's control flow, LLVM expects a target to implement the function `analyzeBranch`, which analyzes the condition and target of a basic block's terminating instruction.

As the function is target-dependent, it has to be generated by the LCB, which currently assumes that branch instructions always follow the RISC-V-style shape, namely two registers being compared, followed by a target.

This assumption fails under AArch64's branch instructions, resulting in compiler crashes as soon as specific optimizations are enabled.

The LCB must analyze an architecture's branch instruction shapes and generate the analyzing function accordingly.

5.4 AArch64 Specification

5.4.1 Multiplication Patterns

AArch64 defines the instruction `MADD` [Lim23a], which allows the multiplication of two registers and adding a third register to the result, storing the final result in a fourth register.

Listing 5.2 shows the definition of the `MADD` instruction in OpenVADL's AArch64 specification. To improve readability, the listing shows the instruction with all macros expanded.

```
1  format MulAddSubFormat: Instr = {
2    sf     [31],
3    op     [30..21, 15],
4    rm     [20..16] : Index,
5    ra     [14..10] : Index,
6    rn     [9..5] : Index,
7    rd     [4..0] : Index}
8
9  instruction MADDW : MulAddSubFormat =
10   let result = VADL::add(W(ra), W(rn) * W(rm)) in
11     W(rd) := result
```

Listing 5.2: OpenVADL definition of the `MADD` instruction.

As described above, two registers are multiplied with each other (`W` indexed by `rm` and `rn`), a third register is added to the result (`W` indexed by `ra`), and the final result is stored in another register (`W` indexed by `rd`).

Listing 5.3 shows the associated patterns generated for the instruction's behavior. Because the instruction covers both addition and multiplication, the associated pattern in TableGen covers these two operations in the same pattern.

```
def : Pat<(add W:$ra, (mul W:$rn, W:$rm)),
          (MADDW W:$ra, W:$rn, W:$rm)>;
```

Listing 5.3: Generated LLVM patterns for the `MADD` instruction defined in listing 5.2.

Because the `MADD` instruction covers multiplication with an additional addition, the LCB never generates sole multiplication patterns. Therefore, a sole multiplication cannot be mapped onto the `MADD` instruction.

To cover the case of a sole multiplication with no addition, OpenVADL's specification defines a pseudo-instruction for multiplication, as can be seen in listing 5.4. The format field `ra` is hardcoded to the decimal value 31, representing the 32nd register of the `W` register, which itself is a zero register. Therefore, this pseudo-instruction covers a single multiplication.

```
pseudo instruction MULW(rd : Index, rn : Index, rm : Index) = {
  MADDW{rd = rd, rn = rn, rm = rm, ra = 0b1'1111}
}
```

Listing 5.4: OpenVADL definition of the `MULW` pseudo-instruction, representing a sole multiplication with a zero-addition.

Although the pseudo-instruction covers the case of a sole multiplication, no appropriate patterns are generated by the LCB, which should recognize the zero addition as an operation with no change to the behavior.

5.4.2 Shift Instructions and Patterns

OpenVADL's AArch64 specification defines typical shift operations, like logical-shift-left (`LSL`) or logical-shift-right (`LSR`), for both the 64-bit register-file `X` and the 32-bit register-file `W`. Both instructions take two registers as operands, representing the value and the shift amount.

In addition to the register-register shift instructions, shift-by-immediate instructions are needed, which are, similarly to section 5.4.1, defined as pseudo-instructions, using the instruction `UBFM` [Lim23b] internally. This can be seen in listing 5.5.

```
pseudo instruction LSLIX(rd : Index, rn : Index, shift : Bits6) = {
  UBFMX{
    rd = rd,
    rn = rn,
    immr = -shift,
```

```
6       imms = (Size::XSize - 1) - shift
7     }
8   }
```

Listing 5.5: OpenVADL definition of the `LSL` instruction, representing a logical-shift-left with two registers.

Although the pseudo-instruction implements a logical-left-shift-by-immediate, the LCB does not correctly recognize the operation and consequently does not generate patterns for LLVM-IR's `lsl` instruction.

5.4.3 Embench Performance Evaluation

The OpenVADL project provides a continuous benchmarking dashboard called Resselpark [Opee], which provides performance evaluation for the OpenVADL frontend, Instruction Set Simulator (ISS) and LCB, using the benchmark Embench [Emb]. Other papers working on OpenVADL also make use of Embench to evaluate the LCB's performance compared to the upstream compilers [Gis] [Per25] [FHH+25b].

While this work extended the LCB, problems mentioned in this chapter prevent most of the benchmark's test files from compiling for the AArch64 specification yet. Making the compilation and execution of Embench for the AArch64 specification possible would be a big step in bringing the LCB and AArch64 specification closer to being production-ready.

CHAPTER 6

Conclusion

OpenVADL's LCB was previously shown to be capable of handling a variety of specifications for different architectures. Nonetheless, each new specification highlights the LCB's abilities while also showcasing shortcomings and new challenges.

This work identified and fixed four such shortcomings that became apparent while generating a compiler for OpenVADL's specification of a subset of the AArch64 architecture. These concerned the handling of the register and subregister hierarchy, a type mismatch between immediate operands in OpenVADL and LLVM, the generation of redundant predicate functions, and the lack of classification for certain machine instructions. The evaluation showed that most test files, which previously failed to compile or executed incorrectly, now compile and execute correctly.

As OpenVADL currently ships only a subset of the AArch64 architecture, and the fixed defects concerned the LCB in general, these fixes are a step towards supporting more complete AArch64 specifications in the future.

Overview of Generative AI Tools Used

Claude Sonnet 5 [Ant] was used in this thesis to find inconsistencies in the writing format, including miscapitalized headings and inconsistent use of text styling. Furthermore, Claude Sonnet 5 assisted in the implementation of LaTeX [LaT] constructs, including tables and graphics, while the actual content of these constructs was not generated.

Additionally, Grammarly [Gra] was used to find and correct grammatical and spelling errors.

No text passages were generated by AI tools.

List of Figures

List of Tables

Acronyms

ABI Application Binary Interface. 9

DAG Directed Acyclic Graph. 5, 6, 9

GCB Generic Compiler Backend. 9

ISA Instruction Set Architecture. 9–12

ISS Instruction Set Simulator. 37

LCB LLVM Compiler Backend. xi, xiii, 9, 11, 13, 15, 19–23, 25, 31, 33–37, 39

LLVM-IR LLVM Intermediate Representation. 3–6, 21, 23, 34, 37, 43

MiA Microarchitecture. 9

PDL Processor Description Language. xi, xiii, 1, 9

SSA Static Single Assignment. 3

VADL Vienna Architecture Description Language. 1, 3, 9–13, 16, 18, 20, 21

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[Ant] Anthropic. Claude Sonnet 5. Accessed: 2026-09-08. URL: https://platform.claude.com/docs/en/models/sonnet-5/overview.

[AW] David A. Patterson Krste Asanovic Andrew Waterman, Yunsup Lee. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Accessed: 2026-08-12. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/Archive/EECS-2011-62.pdf.

[CML08] Anupam Chattopadhyay, Heinrich Meyr, and Rainer Leupers. Processor Description Languages. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 95–132. Morgan Kaufmann, Burlington, 2008. URL: https://www.sciencedirect.com/science/article/pii/B9780123742872500082, doi:10.1016/B978-012374287-2.50008-2.

[Cor24] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2024. Order Number: 325462. Accessed: 2026-08-12. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[Emb] Embench. Embench. Accessed: 2026-09-03. URL: https://www.embench.org/.

[FHH+25a] Florian Freitag, Linus Halder, Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Benjamin Kasper, Niklas Mischkulnig, Michael Nestler, Philipp Paulweber, Kevin Per, Matthias Raschhofer, Alexander Ripar, Tobias Schwarzinger, Johannes Zottele, and Andreas Krall. The Vienna Architecture Description Language, 2025. URL: https://arxiv.org/abs/2402.09087, arXiv:2402.09087.

[FHH+25b] 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 Sven Tomforde, Christian Krupitzer, Stéphane Vialle, Estela Suarez, and Thilo Pionteck, editors, *Architecture of Computing Systems*, pages 156–171, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-032-03281-2_11.

[Gis] Johannes Gisy. Optimizing the OpenVADL compiler generator. Bachelor's thesis, Technische Universität Wien, Wien, 2026. Accessed: 2026-08-12. URL: https://www.complang.tuwien.ac.at/vadl/papers/GisyFinal.pdf.

[Gra] Grammarly. Grammarly. Accessed: 2026-09-08. URL: https://app.grammarly.com/.

[HP94a] John L. Hennessy and David A. Patterson. 3 - Instructions: Language of the Machine. In *Computer Organization and Design*, pages 92–164. Morgan Kaufmann, 1994. URL: https://www.sciencedirect.com/science/article/pii/B9781483207759500117, doi:10.1016/B978-1-4832-0775-9.50011-7.

[HP94b] John L. Hennessy and David A. Patterson. 4 - Arithmetic for Computers. In *Computer Organization and Design*, pages 166–266. Morgan Kaufmann, 1994. URL: https://www.sciencedirect.com/science/article/pii/B9781483207759500129, doi:10.1016/B978-1-4832-0775-9.50012-9.

[LA04] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.

[LaT] LaTex. LaTeX - A document preparation system. Accessed: 2026-09-08. URL: https://www.latex-project.org/.

[Lim] Arm Limited. Documentation – Arm Developer. Accessed: 2026-08-19. URL: https://support.arm.com/documentation/ddi0487/mc/-Part-D-The-AArch64-System-Level-Architecture/-Chapter-D1-The-AArch64-System-Level-Programmers--Model/-D1-5-Process-state--PSTATE/-D1-5-1-PSTATE-fields-that-are-meaningful-in-AArch64-state.

[Lim23a] Arm Limited. AArch64 MADD Instruction, 1 2023. Accessed: 2026-09-08. URL: https://support.arm.com/documentation/dui0801/l/A64-General-Instructions/MADD--A64-.

[Lim23b] Arm Limited. AArch64 UBFM Instruction, 1 2023. Accessed: 2026-09-08. URL: https://support.arm.com/documentation/dui0801/l/A64-General-Instructions/UBFM--A64-.

[Lim25] Arm Limited. Registers in AArch64, 9 2025. Accessed: 2026-08-14. URL: https://support.arm.com/documentation/102374/0103/Registers-in-AArch64---general-purpose-registers.

[Lim26a] Arm Limited. Arm A-profile A64 Instruction Set Architecture, 6 2026. Accessed: 2026-08-19. URL: https://support.arm.com/documentation/ddi0602/2026-06/Base-Instructions/B-cond--Branch-conditionally-?lang=en.

[Lim26b] Arm Limited. Arm A-profile A64 Instruction Set Architecture, 6 2026. Accessed: 2026-08-19. URL: https://support.arm.com/documentation/ddi0602/2026-06/Base-Instructions/CSEL--Conditional-select-?lang=en.

[Lim26c] Arm Limited. Arm A-profile A64 Instruction Set Architecture, 6 2026. Accessed: 2026-08-12. URL: https://support.arm.com/documentation/ddi0602/2026-06/Base-Instructions/STRB--register---Store-register-byte--register--.

[Lim26d] Arm Limited. Arm A-profile A64 Instruction Set Architecture, 6 2026. Accessed: 2026-08-12. URL: https://support.arm.com/documentation/ddi0602/2026-06/Base-Instructions/BLR--Branch-with-link-to-register-.

[Lim26e] Arm Limited. Arm A-profile Architecture Registers, 6 2026. Accessed: 2026-08-12. URL: https://support.arm.com/documentation/ddi0601/2026-06/AArch64-Registers/NZCV--Condition-Flags.

[Muc98] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[Opea] OpenVADL. openvadl/sys/aarch64/aarch64-abi.vadl at 717b1c396e5af6e82355b8a0d40cf7a97a6869d5. Accessed: 2026-08-19. URL: https://github.com/OpenVADL/openvadl/blob/717b1c396e5af6e82355b8a0d40cf7a97a6869d5/sys/aarch64/aarch64-abi.vadl.

[Opeb] OpenVADL. openvadl/sys/aarch64/Xaarch64-abi.vadl at 717b1c396e5af6e82355b8a0d40cf7a97a6869d5. Accessed: 2026-08-19. URL: https://github.com/OpenVADL/openvadl/blob/717b1c396e5af6e82355b8a0d40cf7a97a6869d5/sys/aarch64/aarch64-abi.vadl.

[Opec] OpenVADL. openvadl/vadl/test/resources/llvm/riscv/qemu_libc at 717b1c396e5af6e82355b8a0d40cf7a97a6869d5 · OpenVADL/openvadl · GitHub. Accessed: 2026-08-19. URL: https://github.com/OpenVADL/openvadl/tree/717b1c396e5af6e82355b8a0d40cf7a97a6869d5/vadl/test/resources/llvm/riscv/qemu_nolibc.

[Oped] OpenVADL. openvadl/vadl/test/resources/llvm/riscv/qemu_nolibc at 717b1c396e5af6e82355b8a0d40cf7a97a6869d5 · OpenVADL/openvadl · GitHub. Accessed: 2026-08-19. URL: https://github.com/OpenVADL/openvadl/tree/717b1c396e5af6e82355b8a0d40cf7a97a6869d5/vadl/test/resources/llvm/riscv/qemu_nolibc.

[Opee] OpenVADL. Resselpark. Accessed: 2026-09-02. URL: https://openvadl.github.io/resselpark/?backend=frontend&suite=native&isa=aarch32&key=__all__.

[Per25] Kevin Per. LLVM compiler generator in OpenVADL. Master's thesis, Technische Universität Wien, Wien, 2025. doi:10.34726/hss.2025.122088.

[Proa] LLVM Project. A Beginners Guide to SelectionDAG. Accessed 2026-08-13. URL: https://llvm.org/devmtg/2024-10/slides/tutorial/MacLean-Fargnoli-ABeginnersGuide-to-SelectionDAG.pdf.

[Prob] LLVM Project. Getting Started with the LLVM System. Accessed: 2026-08-13. URL: https://llvm.org/docs/GettingStarted.html.

[Proc] LLVM Project. LLVM: include/llvm/CodeGen/GlobalISel/GIMatchTableExecutor.h Source File. Accessed: 2026-08-12. URL: https://llvm.org/doxygen/GIMatchTableExecutor_8h_source.html#L62.

[Prod] LLVM Project. LLVM Language Reference Manual. Accessed: 2026-08-12. URL: https://llvm.org/docs/LangRef.html.

[Proe] LLVM Project. llvm-project/llvm/include/llvm/CodeGen/ISDOpcodes.h. Accessed: 2026-08-12. URL: https://github.com/llvm/llvm-project/blob/llvmorg-19.1.7/llvm/include/llvm/CodeGen/ISDOpcodes.h.

[Prof] LLVM Project. llvm-project/llvm/lib/CodeGen/BranchFolding.cpp. Accessed: 2026-09-08. URL: https://github.com/llvm/llvm-project/blob/llvmorg-19.1.7/llvm/lib/CodeGen/BranchFolding.cpp.

[Prog] LLVM Project. llvm-project/llvm/lib/CodeGen/MachineBlockPlacement.cpp. Accessed: 2026-09-08. URL: https://github.com/llvm/llvm-project/blob/llvmorg-19.1.7/llvm/lib/CodeGen/MachineBlockPlacement.cpp.

[Proh] LLVM Project. llvm-project/llvm/lib/CodeGen/TailDuplication.cpp. Accessed: 2026-09-08. URL: https://github.com/llvm/llvm-project/blob/llvmorg-19.1.7/llvm/lib/CodeGen/TailDuplication.cpp.

[Proi] LLVM Project. llvm-project/llvm/utils/TableGen/Common/CodeGenDAGPatterns.cpp. Accessed: 2026-08-12. URL: https://github.com/llvm/llvm-project/blob/llvmorg-19.1.7/llvm/utils/TableGen/Common/CodeGenDAGPatterns.cpp.

[Proj] LLVM Project. TableGen Overview - LLVM. Accessed: 2026-08-13. URL: https://llvm.org/docs/TableGen/.

[Prok] LLVM Project. Writing an LLVM Backend. Accessed: 2026-08-12. URL: https://llvm.org/docs/WritingAnLLVMBackend.html.

[RT22] Fabrice Rastello and Florent Bouchez Tichadou, editors. *SSA-based Compiler Design*. Springer, Cham, 1 edition, 2022. doi:10.1007/978-3-030-80515-9.