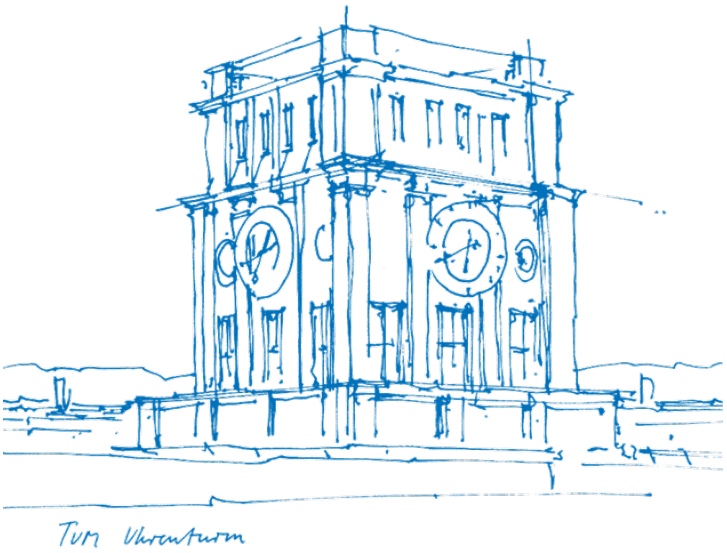
School of Computation, Information and Technology – Informatics
Technical University of Munich

Automatic Integration of Custom RISC-V Instructions into LLVM Toolchain

Bachelor's Thesis in Informatics

Mathis Salmen



School of Computation, Information and Technology – Informatics
Technical University of Munich

Automatic Integration of Custom RISC-V Instructions into LLVM Toolchain

Automatisierte Integration von benutzerdefinierten RISC-V Instruktionen in LLVM Toolchain

Bachelor's Thesis in Informatics

Mathis Salmen

Thesis for the attainment of the academic degree

Bachelor of Science (B.Sc.)

at the School of Computation, Information and Technology of the Technical University of Munich.

Supervisor:

Dr.-Ing. Daniel Müller-Gritschneder

Advisor:

M.Sc. Philipp van Kempen

Submission Date:

Munich, 16.08.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.08.2023 Mathis Salmen

Abstract

The RISC-V instruction set architecture is known for its openness to custom instruction set extensions. Custom extensions allow vendors to add new instructions to RISC-V easily. Many of these extensions are built with performance improvement in mind, often e.g. adding instructions that combine multiple official RISC-V instructions into one. While numerous such extensions have been defined and implemented in processors, compiler support for the generation of their instructions has largely lagged behind. This may be due to the relative ease of adding an instruction to a processor, as opposed to implementing compiler support for emitting an instruction.

In this thesis, we aim to address this mismatch by leveraging LLVM for the generation of its own Instruction Selection Patterns. We present *CoreDSLToLLVM*, a compiler for the conversion of C-style instruction behavior definitions to LLVM Instruction Selection Patterns. While CoreDSLToLLVM generates Instruction Selection Patterns for LLVM, it also uses LLVM internally for both optimization of instruction behavior descriptions and generation of LLVM SelectionDAGs. Instead of being used for instruction selection as intended, the DAGs are output as Instruction Selection Patterns by CoreDSLToLLVM's backend.

For evaluation of CoreDSLToLLVM, we attempt to implement LLVM support for the CORE-V SIMD RISC-V extension, which adds SIMD instructions operating on 8 or 16-bit elements within regular 32-bit integer registers for use in applications such as signal processing and edge machine learning. Given a CoreDSL 2 implementation of the extension, we are able to automatically compile 86% of instructions in the extension to equivalent Instruction Selection Patterns. When using the generated patterns to compile code, the SIMD instructions defined in the extension are successfully emitted by LLVM in appropriate situations. In easily vectorizable workloads using 8 or 16-bit integers, we achieve significant speedups of 2-4x.

Contents

1 Introduction

RISC-V is an open-source instruction set architecture (ISA). The RISC-V standard, maintained by the RISC-V Foundation, is freely available and can be implemented by anyone, even commercially. In the standard and various official extensions, instructions making up the RISC-V instruction set are defined. In addition to these *official* instructions, RISC-V is very open to *custom* instruction set extensions developed by other parties — even reserving space in the instruction set specifically for them. The idea being that anyone can adapt RISC-V to their specific use case by means of custom instructions and other extensions. [1]

Fragmentation A widely raised concern regarding this attitude has been the possibility of fragmentation of RISC-V, as vendors may implement multiple incompatible extensions, possibly even all aiming to address the same issue [2]. Secondarily, compilers often significantly lag behind actual hardware in the implementation of instructions. Many extensions have been implemented, but commonly lack code generation support in common compilers like LLVM [3, 4].

Clearly, the solution to these issues is largely organizational in nature. For example, vendors need to cooperate to avoid the duplicate definition of similar extensions. If an extension is generic enough, it should be officially specified and ratified expeditiously by the RISC-V Foundation, such that a definitive official extension to be used by everyone exists. For the second problem, cooperation between hardware and compiler engineers is needed the ensure timely implementation of newly defined extensions in compilers. To simplify this process, extensions need to be defined with not just hardware in mind, but also compilers.

Compiler Support One *technical* aspect which can be of help in essentially all of the aforementioned issues is the simplification or even automation of integrating new instructions into compilers. One way of achieving this is by generating compiler support for instructions from formal definitions for these instructions. These formal definitions are not unlike regular code, they are also written in a programming language. However, instead of specifying program behavior, the format and behavior of instructions is specified. Ideally, both custom and official instruction set extensions would be redistributed in the form of these formal definitions.

This would allow for much-improved cooperation between compiler and hardware engineers, as compiler engineers would be able to integrate extensions, or even drafts of extensions quickly into compiler toolchains. This can be used for purposes of simple evaluation or more exhaustive design space analysis. While likely not optimal, automatically generated support could also provide a minimal-effort baseline compiler implementation for essentially all extensions. Official extensions will likely always enjoy more optimized, manually implemented compiler support, but this will likely not be the case for niche, custom extensions. Even if all else fails, and a large number of (redundant) extensions do exist, automatic integration into compilers would very least allow for all of them to be used with little effort.

LLVM One may argue that the LLVM Toolchain itself already provides a form of “automatic integration” of custom instructions with its TableGen language for instruction definitions. Unlike in compilers based on simpler paradigms such as pure macro expansion, it is not necessary to alter LLVM’s code generation or

internal logic to implement new instructions. For *most* instructions, we essentially just define the encoding and implement one or multiple Instruction Selection Patterns. [5]

Of course, there are many caveats to this, and it is fundamentally not intended for our goals. Most pressingly, Instruction Selection Patterns are written in a very specialized language not particularly suitable for formal instruction definitions, which need to be generally useful. In addition, the process of writing useful patterns may be very difficult, as any written pattern is largely used *as-is* by LLVM. In other words, there are many ways to express the same computation in a pattern; likely only few of which are useful to LLVM. [5]

CoreDSLToLLVM Thus, instead of implementing instruction support in LLVM by hand, in this thesis, we implement a compiler to automatically generate the required code to implement an instruction in LLVM. This provides much-improved flexibility, as instructions can be described in a much more generic and easily understandable language, namely CoreDSL 2, which has been designed specifically for the purpose of formal instruction definition. While the compiler outputs LLVM Instruction Selection Patterns, it also uses LLVM internally for the generation of these patterns. This significantly reduces implementation effort, and also allows using instructions definitions for purposes other than pattern generation.

1.1 State of the Art

As discussed in the previous section, LLVM SelectionDAG's Instruction Selection Patterns likely represent one of the most widely used technologies providing at least some level of automation for the integration of new instructions.

Beyond this, others have attempted to further simplify the process of adding code generation support for custom instructions. Auler, Centoducatte, and Borin implemented ACCGen in 2013 [6], which generates from compilers from ArchC, a domain-specific language similar to CoreDSL 2. They also generate LLVM patterns but focus on the base implementation of several different ISAs including ARM and MIPS. They also do not prominently use LLVM within the process of pattern generation.

Buchwald, Fried, and Hack [7] describe a fully automatic approach to generate provably correct instruction selection rules from a formal description of x86 integer arithmetic instructions. Their approach is focused on formal correctness and thus requires "a few days" to generate rules. In contrast, our approach does not provide formal correctness, instead relying on LLVM for transformations; but completes generation in a negligible amount of time.

The general distinguishing factors of the work presented here are largely two-fold. First, we do not attempt to generate instruction support for all instructions within arbitrary ISAs. Our focus is on adding support for RISC-V extensions. For base instructions, we utilize existing support within LLVM. Secondarily, we present the novel approach of using LLVM, in particular LLVM SelectionDAG for the generation of its own patterns. This significantly reduces implementation effort, as we essentially just implement an LLVM Frontend and Backend.

1.2 Content of this Thesis

In the following chapter, we explain the necessary prerequisites from RISC-V, LLVM, and CoreDSL 2 for this thesis. Then, we move to the implementation of CoreDSLToLLVM and related tools. After that, we show and analyze the results of the implementations by example. Finally, we conclude the thesis and discuss known limitations and possible future work.

2 Background

2.1 RISC-V

Unless otherwise noted, this section is based on [1].

RISC-V is an open-source instruction set architecture (ISA) that has gained significant traction in recent years. The RISC-V architecture was developed at the University of California, Berkeley starting in 2010, with the intention of providing a free and open standard for designing computer chips in academia and industry.

RISC-V is based on Reduced Instruction Set Computing (RISC) principles, which prioritize simplicity and efficiency in instruction execution. Almost all instructions conform to one of few simple patterns in encoding and behavior, allowing for straightforward decoding and execution.

2.1.1 General

RISC-V defines a 32-bit, 64-bit, and (so far unratified) 128-bit base instruction set. The number of bits, or `XLEN` in RISC-V terminology, specifies the size in bits of an integer register, also called GPR (general purpose register). Within the most common `rv32i` and `rv64i` instruction sets, there are 32 integer registers, namely `x0` to `x31`. However, `x0` is a constant zero register, and cannot be overwritten.[1]

Independent of `XLEN` is the size of an instruction, which is the same for 32-bit, 64-bit, and 128-bit RISC-V processors. The only instruction size a RISC-V processor must support is 32-bit, as all base RISC-V instructions are 32-bit instructions. Optionally, 16-bit *compressed* instructions — commonly "compressed" aliases of regular 32-bit instructions — can be implemented to reduce code size.[2]

In general, most 32-bit RISC-V instructions are in three-operand form. When looking at a particular instruction, its source registers are named `rs1` and `rs2`, while the destination register is named `rd`. For many instructions, the second source operand can be a constant value instead of register `rs2`. In this case, it is referred to as an *immediate operand*, commonly shortened to `imm`.

In this thesis, we exclusively use `rv32i`, the 32-bit, 32-register variant of the RISC-V ISA. We also limit ourselves to 32-bit instructions.

2.1.2 Extensions

One of the notable features of RISC-V is its modular structure. In addition to providing a 32, 64, and 128-bit base instruction set for various applications; encoding space is reserved for official as well as custom RISC-V Extensions providing additional instructions or other features. Official extensions include, among many others, the M-Extension for multiplication and division, or the A-Extension for Atomic Memory Instructions.

[1] In practice, alternative register names such as `zero` for `x0`, `a0` – `a7` for function argument registers or `s0` – `s11` for callee-saved registers are often used instead of plain `x0` – `x31`.

[2] Encoding space is also reserved for larger-than 32-bit instructions (in 16-bit increments), such as 48 or 64-bit instructions. As of now, this encoding space is unused by official RISC-V instructions.

Unofficial (custom) extensions are encouraged by RISC-V as well. These commonly include implementation-specific instructions for configuration or e.g. cache flushes [8]. Of particular importance for this thesis however, are extensions that add additional computational instructions. These commonly aim to improve application performance by providing instructions that perform more complicated or combined computations. Examples of this are integer multiply-accumulate instructions or load/stores which increment the address register after memory access [8].

CORE-V Extension

The CORE-V-Extension is the custom RISC-V extension used extensively as part of this thesis. It was developed alongside the CV32E40P 32-bit RISC-V processor and aims to overcome some of RISC-V's shortcomings for small processor cores [9].

RISC-V itself aims to "[avoid] 'over-architecting' for a particular microarchitecture style". This is a sensible choice, as RISC-V aims to be a generic ISA to be used in many different classes of processors. It does however imply that some useful instructions or other features cannot be included in official the RISC-V ISA, as they are too tailored to a particular processor or microarchitecture.

This is particularly problematic for small, in-order, microcontroller-class RISC-V processors such as CV32-E40P. Unlike larger *application processors*, which can mitigate mismatches between ISA and microarchitecture using techniques such as instruction fusion and superscalar execution; the microarchitecture of small processors is much more bound to the ISA, as these techniques cannot be implemented.

This problem is alleviated by the use of custom extensions for particular processors or microarchitectures, of which the CORE-V-Extension is one example. It defines the following specialized instructions:

- Loads and Stores which increment the address register
- Indexed Loads and Stores with offset and address registers (with and without increment)
- Hardware Loop Instructions to execute sections of code multiple times without loop overhead
- Bit-Manipulation Instructions (extract, insert, . . .)
- General ALU Instructions (min, max, sign/zero extend, average, . . .)
- Branches with immediate comparison operands
- Integer Multiply-Accumulate Instructions
- SIMD Instructions on 32-bit GPRs

In this thesis, we only consider instructions without side effects, i.e. no memory access or branching. In particular, the main focus is on the SIMD instructions defined as part of the CORE-V extensions, or CORE-V SIMD.

2.1.3 SIMD

Generally, instructions such as `add` perform a single operation on scalar operands. Some specialized instructions may even perform multiple operations "at once", such as the aforementioned multiply-accumulate instructions, which perform both multiplication and addition.

SIMD — single instruction, multiple data — instructions go one step further by explicitly performing the same operation for multiple inputs. A common example is an instruction that adds two vectors of n numbers[3]. n independent additions are performed, but a single SIMD instruction suffices, and will likely run at about the same performance as a regular single `add`.

[3] In practice, n will range anywhere from 2 to about 64 (depending on the width of a vector and that of a single element)

This is termed a *lane-wise* SIMD instruction, as the calculation is performed independently for each lane. This type of SIMD instruction is the most common. In addition, some SIMD instruction sets define *horizontal* SIMD instructions, alternatively called *reductions*. An example of this would be an operation that computes the sum of elements within *one* vector, thus reducing the vector to a scalar value.

SIMD instructions are widely implemented in application processors and used for computationally intensive tasks such as computer graphics and machine learning. Examples of common SIMD extensions include various versions of SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) on x86-64 processors, as well as NEON and SVE on ARM processors. [10, 11]

2.1.4 RISC-V Vector Extension

To compete with x86 and ARM, RISC-V also defines a SIMD extension, the *RISC-V Vector Extension* or *RVV*. Unlike the basic RISC-V instruction set, which is based on proven principles, the RVV extension is somewhat of a departure from classical SIMD extensions. In particular, it does not define a constant vector size, unlike e.g. ARM NEON and x86 SSE, which both use 128-bit vectors/registers.[4] [10, 11, 12]

Instead, a vector length is selected at runtime via the `vsetvl` ("set vector length") instruction. This aims to increase flexibility by allowing CPU designers to freely choose a vector length for every particular CPU design, without breaking compatibility with existing binaries. This is in contrast to classical SIMD. For example, the upgrade from x86 SSE (128-bit registers) to x86 AVX (256/512-bit registers) requires, at the very least, recompilation of source code to use new instructions and registers. [12]

RVV thus provides a neat solution for SIMD on application processors, where binary compatibility is of high importance. It does however come at a cost of significantly increased complexity — as compared to classical SIMD — to implement a variable vector length and dedicated vector registers. This is particularly problematic for simple, microcontroller-class processor cores, which do not have the *silicon budget* to implement complex features. They could however greatly benefit from a simple SIMD extension for such tasks as signal processing or edge machine learning.

2.1.5 RISC-V Packed SIMD

To alleviate this deficiency, various parties have defined classical *packed* SIMD extensions for RISC-V as well. These generally do not intent to compete with RVV, but are instead intended to provide a simpler set of SIMD instructions for microcontrollers or small digital signal processors.

Currently, none of these extensions are *ratified*, i.e. officially endorsed, by the RISC-V Foundation. Still, most notable is the RISC-V P Extension proposal, which adds an extensive amount of SIMD and other instructions to speed up various computations [13]. In this paper, we will use the smaller CORE-V SIMD extension, part of the larger CORE-V extension introduced in Section 2.1.2 and implemented in the open source CV32E40P processor [9].

As these packed SIMD extensions are aimed at microcontrollers, the do not define additional register state. Instead, vectors are stored within regular integer registers, thus limiting their size to, in our case, 32 bits. They also do not have the ability to operate on floating-point vectors, only 8 and 16-bit integers are supported [9, 13].

2.1.6 CORE-V SIMD

The CORE-V SIMD Extension operates on vectors of 4x8-bit elements or 2x16-bit elements, packed into regular 32-bit integer registers. As regular integer registers are used, regular load/store instructions are

[4] ARM Scalable Vector Extensions (SVE) notably uses the same approach as RVV.

Mnemonic	Description
`cv.OP.h rd, rs1, rs2`	Execute `OP` for matching 16-bit elements in `rs1`, `rs2`
`cv.OP.h.sc rd, rs1, rs2`	Execute `OP` for 16-bit elements in `rs1`, combined with first element in `rs2`
`cv.OP.h.sc.i rd, rs1, imm6`	Execute `OP` for 16-bit elements in `rs1`, combined with immediate `imm6`
`cv.OP.b rd, rs1, rs2`	Execute `OP` for matching 8-bit elements in `rs1`, `rs2`
`cv.OP.b.sc rd, rs1, rs2`	Execute `OP` for 8-bit elements in `rs1`, combined with first element in `rs2`
`cv.OP.b.sc.i rd, rs1, imm6`	Execute `OP` for 8-bit elements in `rs1`, combined with immediate `imm6`

Table 2.1 CORE-V SIMD Instruction Kinds (in all cases: result in `rd`)

Instruction	funct5 [31:27]	[26]	[25]	[24:20]	[19:15]	funct3 [14:12]	[11:7]	opcode [6:0]
`cv.add.h`	00000	0	0	rs2	rs1	000	rd	1111011
`cv.add.b`	00000	0	0	rs2	rs1	001	rd	1111011
`cv.add.sc.h`	00000	0	0	rs2	rs1	100	rd	1111011
`cv.add.sc.b`	00000	0	0	rs2	rs1	101	rd	1111011
`cv.add.sci.h`	00000	0	imm[0]	imm[5:1]	rs1	110	rd	1111011
`cv.add.sci.b`	00000	0	imm[0]	imm[5:1]	rs1	111	rd	1111011

Table 2.2 CORE-V SIMD Instruction Encoding. [x:y] is an *inclusive* range, e.g. [6:0] is the low 7 bits

used for SIMD as well. No SIMD-specific branches are defined either. Thus, all added instructions are purely computational, with no side effects. Their only effect is storing the computed result value in `rd`.

As shown in Table 2.1, CORE-V SIMD instructions follow a simple RISC-style pattern in their behavior. `Ops` include addition, subtraction, bitwise logic, shifts, and comparisons; as well as more complex operations like dot products. Most of these operations are defined for all of the six formats listed, though some exceptions exist. Some operations may also use the previous value in `rd` as an operand, such as accumulating dot products, which add the result of the dot product to the previous value of `rd`. [9]

All CORE-V SIMD instruction mnemonics are prefixed with `cv.`, and end with the instruction kind as shown in Table 2.1, if multiple kinds are defined. In some environments, the dots in the instruction name are unsupported. In these cases, we use an uppercase name with underscores instead of dots. As such, `CV_ADD_SC_H` is equivalent to `cv.add.sc.h` in this thesis.

Encoding

In their binary encoding, CORE-V SIMD instructions follow the style set by official RISC-V instructions, as shown in Table 2.2. The lowest 7 bits are the *major opcode*, which is the same for all instructions in the CORE-V extension. In particular, it is the *custom-3* major opcode, reserved by RISC-V for custom extensions. The 5-bit fields for `rd`, `rs1` and `rs2` are in the same position as in official RISC-V instructions. For immediate instructions, the `rs2` field is instead used to store 5 bits of the immediate, with the sixth bit being stored in bit 26, which is otherwise set to zero. [9]

For CORE-V SIMD instructions, the particular operation performed is encoded within the `funct3` and `funct5` opcode fields. `funct3` encodes the instruction kind (see Table 2.1). As only six kinds are defined for eight values of `funct3`, two values of `funct3` are unused. `funct5` encodes the actual operation performed, and is thus the same (0) for all instructions in Table 2.2, as all perform an `add`. [9]

2.2 LLVM

LLVM[5] is a toolbox of compiler technologies. It aims to enable compilation of any programming language to any target architecture. Support for additional programming languages can be added by creating *LLVM Frontends*, while target architectures are implemented as *LLVM Backends*. In between Frontend and Backend, programs are kept as *LLVM IR* (intermediate representation) code. LLVM optimizes this code using the LLVM Target-Independent Optimizer. [14]

2.2.1 LLVM IR

Unless otherwise noted, this section is based on [15].

LLVM IR is a compiler intermediate representation, a common format used by LLVM for the generic representation of programs. It also serves as the input language of the LLVM system. The first step of compilation with LLVM typically is the conversion of the program to LLVM IR.

```
uint foo (uint a, uint b, uint c)
{
  return a + b * c;
}
```

Listing 2.1 Definition of `foo` in C

```
define dso_local i32 @foo
    (i32 noundef %a, i32 noundef %b, i32
        noundef %c) #0
{
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %c.addr = alloca i32, align 4
  store i32 %a, ptr %a.addr, align 4
  store i32 %b, ptr %b.addr, align 4
  store i32 %c, ptr %c.addr, align 4
  %0 = load i32, ptr %a.addr, align 4
  %1 = load i32, ptr %b.addr, align 4
  %2 = load i32, ptr %c.addr, align 4
  %3 = mul i32 %1, %2
  %4 = add i32 %0, %3
  ret i32 %4
}
```

Listing 2.2 LLVM IR for `foo` before optimization, as generated by Clang, the LLVM Frontend for C

```
define dso_local i32 @foo
    (i32 noundef %a, i32 noundef %b, i32
        noundef %c) #0
{
entry:
  %0 = mul i32 %c, %b
  %1 = add i32 %0, %a
  ret i32 %1
}
```

Listing 2.3 LLVM IR for `foo` after optimization

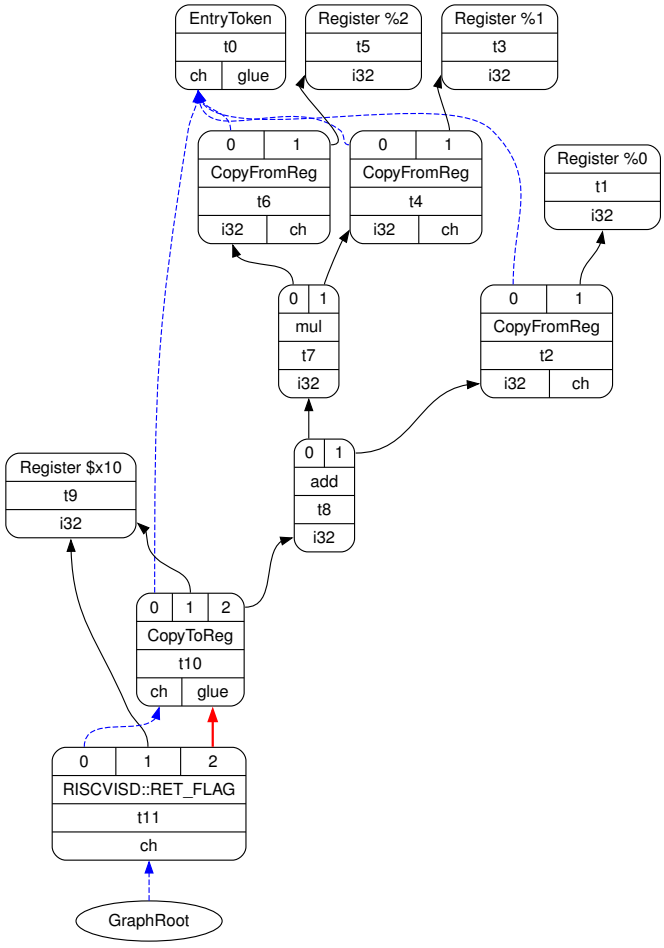


Figure 2.1 SelectionDAG for `foo`, generated from optimized LLVM IR in Listing 2.3

[5] Formerly low-level virtual machine, now no longer an initialism.

As can be seen in Listing 2.2, LLVM IR is superficially reminiscent of RISC-style assembly language, being a list of generic operations, similar to assembly instructions.[6] As LLVM IR is not intended to be used as actual machine code however, it is not subject to some of its limitations. For instance, the number of registers that can be used is not limited. In fact, LLVM IR uses *static single-assignment* (SSA) form, where every operation's result is placed into an exclusive "register", which only ever contains that result and cannot be overwritten.

Also unlike common assembly languages, LLVM IR is strictly and generically typed. For every instance of an operation within LLVM IR, a type is specified, with which the operation is performed. For typical operations such as `add` and `mul`, both input and output values conform to this type. `add` and `mul` thus truncate the result, as is typical in low-level code. For conversion between types, explicit `trunc`, `sext` and `zext` operations exist, which truncate, sign-extend and zero-extend values, respectively. For example, to perform a non-truncating addition or multiplication, the input values are first sign or zero-extended to the desired result type, then the operation is performed with that type.

Within LLVM IR syntax, integer types are represented as `in`, where `n` is the width in bits. In addition to typical types like `i8`, `i16` and `i32`, types like `i1` (boolean), `i42` or `i9999` are supported as well. There is no distinction made between signed and unsigned integers within types themselves. The distinction is made by operations affected by signed-ness, such as the previously mentioned sign-extend and zero-extend. Other types include IEEE floating-point types, and `ptr`, an opaque pointer type (the type of the value pointed to is defined by the operation accessing the pointer).

In addition to these basic types, and importantly for this thesis, LLVM also has built-in support for *vector* types. For example, the type `<4 x i8>` is a vector of four 8-bit integers. Another shorthand syntax often used is in the form of `v4i8`, as in "vector of 4 elements of `i8`". When operations such as `add` or `mul` are run on vector types, the calculation is performed lane-wise, or independently for each of the vector's lanes. For vector-specific operations, such as element extraction or horizontal reduction operations, built-in *intrinsic functions* are provided. In practice, these intrinsics are used in a very similar manner to regular operations.

2.2.2 LLVM Frontend

The frontend is responsible for parsing a programming language and converting the program to equivalent LLVM IR. As the LLVM IR will be analyzed and optimized in the following steps, relatively little care has to be taken with regard to program efficiency at this stage — the input program can just be converted to LLVM IR in a naive manner. One of the most widely used LLVM Frontends is *Clang*, which converts C, C++ and Objective-C to LLVM IR [14].

While LLVM IR can be generated manually as either text or bytecode, the LLVM C++ library is typically used for constructing programs in LLVM IR. Thus, instead of manually generating e.g."`%2 = add i32 %0, %1`" as text or equivalent bytecode, a library function such as `llvm::IRBuilder::CreateAdd(Value* LHS, Value* RHS)` is used.

2.2.3 LLVM Optimizer

The LLVM Optimizer optimizes LLVM IR previously generated by the frontend. Optimizations performed at this stage are (relatively) target-independent, therefore all targets can use the common implementation contained in the LLVM Optimizer. Examples of performed optimizations are:

- Dead Code Elimination (DCE): Remove code that does not affect program result
- Constant Propagation: Pre-compute compile-time constant values

[6] Note that, throughout this paper, we use *operation* to denote generic "instructions" such as those in LLVM IR (or later, SelectionDAG); while *instruction* is used for specific machine (typically RISC-V) instructions.

- Loop Unrolling: Duplicate loop bodies multiple times to reduce the number of taken branches
- Auto Vectorization: Attempt to parallelize loop iterations or independent scalar operations

These and many more optimizations are run on LLVM IR as *passes*. Passes run one after the other, each one possibly modifying the program's LLVM IR to apply its particular transformation. [16]

Shown in Listing 2.3 is the result of optimizing the LLVM IR in Listing 2.2, using the `opt` tool included with LLVM. In this simple example, redundant stores to and loads from the stack have been optimized out, leaving just multiply, add and return.

2.2.4 LLVM Backend

Unless otherwise noted, this section is based on [5].

Generally, backends convert LLVM IR to target-specific machine code while also performing target-specific optimizations. As LLVM IR — compared to machine code — is still a moderately high-level program representation, it is typically not converted to machine code directly. Instead, LLVM provides various further and lower-level intermediate program formats to simplify backend implementation.

SelectionDAG

The most notable of these is LLVM SelectionDAG (directed acyclic graph). Unlike LLVM IR and most programming languages, the SelectionDAG is not a linear sequence of operations. Instead, the flow of data through operations is represented as a directed acyclic graph for all of the program's *basic blocks* — a basic block being a sequence of operations that ends with a branch (terminator) but does not contain any branches within the rest of its code.

SelectionDAG is mainly used to analyze paths of data flow through operations. Using a linear sequence of operations would merely increase ambiguity, as the same data flow can be represented using different instruction orderings unless every instruction depends on the result of its immediate predecessor. Some non-dataflow dependencies can still exist, such as a certain store being performed before a certain load. This may be because of a direct memory dependency between them, for example, if the store writes to the same address the load reads from.[7] Alternatively, memory accesses might be marked as `volatile`, thus re-ordering is not allowed, as is common for e.g. memory mapped IO.

The SelectionDAG generated from the previous LLVM IR example is shown in Figure 2.1. In the graph, values are first copied from function argument registers.[8] Using these, the computation is performed. Then, the result is copied to the result register, and the function returns.

In the graph, arrows represent dependencies, for example, the `add` depends on the result of `mul`. The arrow is thus in the *opposite* direction of data flow. The blue and red arrows represent other types of dependencies. For example, the write to the return value register must happen before the function returns, even though no direct dataflow dependency exists between the copy and return operations.

Construction and Usage of SelectionDAG

One of the first steps in a SelectionDAG LLVM backend is the conversion of LLVM IR to SelectionDAG. This conversion is generic and can thus be performed by LLVM, it does not have to be implemented by backend developers. After conversion to SelectionDAG, various optimizations and legalizations are performed. During legalization, operations on types marked as unsupported, or operations otherwise

[7] This may theoretically be classified as a dataflow dependency as well, but is not tracked as such in SelectionDAG.

[8] These redundant copies in the graph will very likely be resolved during register allocation. This is done by mapping intermediate values to the register they originate from, thus removing the need for an explicit copy (unless the original value is still needed).

determined to be unsupported are removed, and replaced by supported operations on supported types. These transformations are largely performed by generic LLVM code as well, backends usually merely pass their parameters. However, manual intervention by backend developers is sometimes necessary for target-specific optimizations or legalizations.

After optimization and removal of unsupported operations, the DAG is ready for *Instruction Selection*. In this step, all generic operations in the DAG are replaced with real machine instructions. This is achieved by means of tree pattern matching, where sequences of one or more generic operations are replaced by one or more target-specific instructions.

These steps, and the steps that follow to generate machine code, can roughly be classified as follows:

1. **DAG Construction**: Convert all basic blocks from LLVM IR to SelectionDAGs. At this point, operations in the SelectionDAGs are generic and target-independent. The following steps run for all SelectionDAGs.
2. **DAG Combine**: Perform low-level optimizations on the SelectionDAG.
3. **Legalize**: Eliminate unsupported types and operations by expanding them to supported operations. On RISC-V (without extensions) for example, a `sign_extend` is converted to a shift left, followed by an arithmetic shift right.
4. **Instruction Selection**: Greedily match machine instructions to sections of the DAG, replacing all previous generic operations. This process is also called *DAG covering*, as all operations in the DAG are "covered" with machine instructions.
5. **Register Allocation**: Allocate hardware registers for intermediate values while respecting register constraints, such as fixed destination registers, or function argument and return value registers.
6. **Instruction Scheduling**: Convert the SelectionDAG back to a linear sequence of instructions, while ordering instructions to avoid pipeline stalls.
7. **Emit Machine Code**

Of most importance for this thesis is the *Instruction Selection* step, as this is where custom instructions can be generated to improve program performance. Additionally, the immediately preceding *Legalize* step is essential as well, as it sets the foundation for Instruction Selection. We do not intervene in any of the other steps in this thesis, except possibly for minor adjustments. In general, LLVM's or the RISC-V LLVM backend's implementation of these is used largely unmodified.

2.2.5 Instruction Selection Patterns

Within the Instruction Selection step, to generate instructions for DAGs of generic operations, LLVM makes use of pattern matching. In particular, we define patterns for machine instructions (or sequences of them), in which the operation performed is described in terms of SelectionDAG's generic operations. During Instruction Selection, these patterns are matched to subsections of the generic SelectionDAG. Matched sections can then be replaced by the corresponding machine instruction(s).

These *Instruction Selection Patterns* are typically written in the specialized programming language TableGen. TableGen is an LLVM-specific language developed to allow for the description of objects in a succinct, high-level format. In addition to Instruction Selection Patterns, it is used for many applications within LLVM where large tables or other structures have to be generated from ideally little input code. [17]

Listing 2.4 shows an example of an Instruction Selection Pattern in TableGen. Both the pattern and machine instructions to replace it are represented as a tree, with the root being the result value. In this case, the target-specific instruction to be generated is a 4 x 8-bit SIMD addition, i.e. all four elements of `rs1` are added to matching elements in `rs2` (see Table 2.1), with the results stored in `rd`.

```
def : Pat<
    // The pattern of generic operations to match
    (add PulpV4:$rs1, PulpV4:$rs2),

    // The target-specific instruction(s) to generate if a match is found
    (CV_ADD_B PulpV4:$rs1, PulpV4:$rs2)>;
```

Listing 2.4 TableGen Instruction Selection Pattern for cv.add.b

```
def : Pat<
    // The pattern of generic operations to match ("v4i8" is a type hint, not an operation)
    (add PulpV4:$rs1, (v4i8 (splat_vector GPR:$rs2))),

    // The target-specific instruction(s) to generate if a match is found
    (CV_ADD_SC_B PulpV4:$rs1, GPR:$rs2)>;
```

Listing 2.5 TableGen Instruction Selection Pattern for cv.add.sc.b

In this case, the generic representation of the computation, i.e. the pattern to match, corresponds one-to-one to the target-specific instruction. Namely, the generic representation is single `add` operation, which performs the same computation as the target-specific `cv.add.b` instruction. As such, during instruction selection, `add` may be replaced by `cv.add.b` for type `v4i8`.

A slightly more complex pattern is shown in Listing 2.5. In this case, the instruction does not merely perform a simple `add`. Instead, the first element of `rs2` is added to all elements of `rs1` (see Table 2.1), with the result once again in `rd`. This is implemented in the pattern by using the `splat_vector` operation, which converts a single value to a vector containing n copies of said value, in this case 4. For the pattern to match, the result of `splat_vector` must then be used by an `add` operation.

Note that, while `splat_vector` is used as the right-hand side for the `add`, the pattern will also be able to match additions with `splat_vector` on the left, as the TableGen instruction pattern backend is aware of the commutativity of addition, and matches both cases.

While we could implement two patterns for *just* `add` and *just* `splat_vector` individually, the second pattern allows us to emit a single instruction, which performs both of these operations. However, we *must* also define patterns for individual `add` and `splat_vector` operations, as we generally cannot guarantee that an `add` will always use a `splat_vector` as one of its operands.

Exclusive and Composite Patterns

This implies that patterns of more than one operation — *composite* patterns — are strictly optional. This is because an *exclusive* pattern only containing the operation must exist for every operation, as we must be able to match something even if the operation is used in isolation.

In Listing 2.4 for example, `cv.add.b` provides an exclusive pattern for the operation `add` with type `v4i8`. Given this pattern, we can safely match any `add`. The same is not true for `splat_vector` in Listing 2.5 however. While `splat_vector` is used in the pattern for `cv.add.sc.b`, this is a composite pattern — it only matches if `splat_vector` is used as an operand for an addition. To safely use `splat_vector`, we must also define an exclusive pattern of just `splat_vector`. In this case, `splat_vector` could be implemented as a `cv.add.sc.b` where the `rs1` is set to zero using the zero register.

While composite patterns are not required for successful compilation, they are very important for generating high-performance code with LLVM. This is because they form the backbone of low-level target-specific optimization in LLVM, as they allow multiple of SelectionDAG's generic operations to be expressed as one instruction. This is a problem historical code-generation techniques like macro expansion have had issues

with, as a single generic operation would usually be expanded into *one or more* machine instructions [18, Chapter 11.2].

2.2.6 Legalization

If we are unable to define an exclusive pattern for a certain generic operation — say our target does not support the operation, and there is no reasonable way to emulate it — said operation must not occur in the DAG when selecting instructions, as selection would fail. In LLVM, the *Legalizer* is responsible for removing these unsupported operations. Before attempting to match patterns to generate machine instructions, the SelectionDAG is *legalized*. During this process, all operations and types which are unsupported or *illegal* are eliminated by replacing them with equivalent legal alternatives.

In particular, the *Legalize Action* is defined for every tuple of `(type, operation)`, and is set to one of four actions when initializing the backend:

- **Legal**: The operation is legal, no action.
- **Promote**: The operation is supported for an equally sized or larger type. Perform the operation as that type. A common example is operations on 8 or 16-bit integers, which use 32-bit instructions on many RISC architectures, including RISC-V.
- **Expand**: The operation is split and performed as multiple operations of a smaller type. A common example is 64-bit operations on 32-bit targets, which are often split into two 32-bit operations.
- **Custom**: Custom C++ code decides if and how to legalize the operation. This is used if the legality of an operation does not just depend on its type — for example, an operation might only be supported if a particular argument is constant. It can also be used for expanding operations which LLVM does not support expanding itself.

Correct legalization is very important, as every operation that remains after legalization must be able to match with at least one Instruction Selection Pattern, or be converted to target-specific instructions by means of some custom mechanism. If neither of these are provided and the operation is generated, a *Match Failure* occurs, which causes the compiler to crash.

As mentioned previously, the mere occurrence of an operation in a composite pattern is not sufficient, as the rest of the pattern might not match. LLVM has no ability to adjust the DAG to “make a pattern fit”, even if that would be the only way to find a match. Thus, an exclusive pattern is required for every legal `(type, operation)` tuple.

2.2.7 Instruction Encoding

While patterns define the behavior and operands of an instruction, we have yet to define its actual format — both within binary machine code, as well as in human-readable assembler input or disassembler output. Like patterns, the format of instructions is defined within TableGen. Every instruction defines an instance of the base *Instruction* class, in which its format and fields are specified.

In Listing 2.6, the encoding for the `cv.add.b` instruction from earlier is defined. Instead of inheriting from the `Instruction` class directly, we use the `RVInst` class provided by the RISC-V backend, which is itself an instance of `Instruction` specialized for RISC-V.

As expected, we specify all used fields within the 32-bit RISC-V instruction word as defined by the CORE-V instruction encoding (see Table 2.2). Most fields are various opcodes to specify the operation, which are all constant. In this case, variable fields comprise of the two register index fields for the operand register

```
def CV_ADD_SCI_B: RVInst<                    // Instance of RVInst (of RISC-V backend)
       (outs PulpV4:$rd),                    // Output Values
       (ins PulpV4:$rs1, simm6:$imm),        // Input Values
       "cv.add.sci.b", "$rd, $rs1, $imm",    // Assembly Strings (name and operands)
       [], InstFormatOther> {

   // Parameters of an instance of the instruction
   // In this case, the register ID fields (5 bits for x0-x31),
   // and an immediate field (6-bit signed for -32 to 31)
   // These need to have the same names as ins and outs.
   bits<6> imm;
   bits<5> rs1;
   bits<5> rd;

   let Inst{31-27} = 0x0;
   let Inst{26-26} = 0x0;
   let Inst{25-25} = imm{0-0};
   let Inst{24-20} = imm{5-1};
   let Inst{19-15} = rs1{4-0};
   let Inst{14-12} = 0x7;
   let Inst{11-7} = rd{4-0};

   // In RISC-V, the major opcode is the lower 7 instruction bits,
   // in this case "custom-3" for custom 32-bit length instructions.
   let Opcode = 0x7b;
}
```

Listing 2.6 The instruction class instance for cv.add.b, defining encoding and assembly string

`rs1`, as well as the destination register `rd`; and the a 6-bit immediate field. As these fields differ between instances of the instruction, member variables of type `bits<N>` are defined to store every instance's concrete values.

The human-readable assembler format of the instruction is given in two parts. First, the instruction name, in this case, `"cv.add.sci.b"` with which the instruction can be used in assembly files. After that, the format of the instruction's operands is given, in this case, the name of `rd` followed by the names of `rs1` and numeric value of the immediate, separated by commas.

In addition to this format-related information, we also define the instruction's input and output value types. The instruction has one output value and one input value of type `PulpV4`, where `PulpV4` is a register type defined to store a `v4i8` vector. A similar type `PulpV2` exists for `v2i16` vectors. Both of these types refer to the same general-purpose integer registers, but are used separately within LLVM to specify the type currently stored. The immediate value on the other hand is not a vector, instead it is defined to be a `simm6`, a two's complement signed 6-bit immediate, thus ranging from -32 to 31.

This input and output information is used for linking the Instruction Selection Patterns presented earlier to the instruction's fields. After a pattern is matched and registers are allocated, a finished machine instruction can be constructed by setting the instruction's fields to the values same values as in the DAG.

It should also be noted that, owing to TableGen, these instruction definitions are very flexible. If multiple instructions share a similar encoding for example, further subclasses of `RVInst` can be defined to reduce duplicate code. For automatically generated descriptions however, definitions can be simple and verbose, like in Listing 2.6, as to be easy to generate programmatically.

```
CV_ADD_B {
  encoding:
    5'b00000 :: 1'b0 :: 1'b0 :: rs2[4:0] :: rs1[4:0] :: 3'b001 :: rd[4:0] :: 7'b1111011;
  assembly: "{name(rd)}, {name(rs1)}, {name(rs2)}";
  behavior: {
    // Do not attempt to overwrite the zero register
    if (rd != 0) {
      X[rd][ 7: 0] = (X[rs1][ 7: 0] + X[rs2][ 7: 0])[7:0];
      X[rd][15: 8] = (X[rs1][15: 8] + X[rs2][15: 8])[7:0];
      X[rd][23:16] = (X[rs1][23:16] + X[rs2][23:16])[7:0];
      X[rd][31:24] = (X[rs1][31:24] + X[rs2][31:24])[7:0];
    }
  }
}
```

Listing 2.7 The definition of the cv.add.b instruction in CoreDSL 2.

2.3 CoreDSL 2

Unless noted otherwise, this section is based on [19].

CoreDSL 2 is a domain-specific programming language used to describe the behavior of processors at a high level. The state of a processor, as well as how instructions modify this state can be defined in CoreDSL 2. It is intended to be used for purposes of simulation, verification, and formal specification of ISAs. Unlike hardware-description languages (HDLs) such as Verilog, CoreDSL 2 is explicitly not indented to describe the implementation of a processor, and can thus remain much more concise.

In this thesis, we exclusively focus on instruction definitions in CoreDSL 2. We do not make use of CoreDSL 2's other features, such as the definition of architectural state. This is possible as the CORE-V extension instructions covered in this thesis do not define any additional state. As such, we simply use existing CoreDSL 2 RISC-V infrastructure for aspects such as the definition of the 32 integer registers.

2.3.1 Instruction Definitions

As shown in Listing 2.7, CoreDSL 2 uses a mixture of C, Verilog and custom syntax for the definition of an instruction. The instruction definition begins with the instruction's identifier, in this case `CV_ADD_B`, and is then split into three sections. Namely, *encoding*, *assembly* and *behavior*. As they serve very similar purposes, these sections parallel the LLVM TableGen constructs described earlier in Sections 2.2.5 and 2.2.7, albeit at a higher level of abstraction.

Encoding

The instruction encoding is defined as a long concatenation of the instruction's various fields. The bitwise concatenation operator `::` defined by CoreDSL 2 is used for this purpose. Just as in TableGen, most fields are constant and can thus be defined with a literal value. The literals use Verilog syntax:

```
<length in bits>'<b(inary)|o(ctal)|d(ecimal)|h(ex)><s if signed><value>
```

Non-constant fields, i.e. register indices and immediates are defined in the encoding specifier as well. Their length is indicated as is done in Verilog, with the index of the highest bit followed by that of the lowest, which is 0. Note that the total length in bits of the field is one more than the highest index, as zero-based indexing is used.

Assembly

The operand string is almost identical to the one in TableGen, the only difference being that we explicitly specify whether to print a numerical value (e.g. for an immediate) or a register name. Unlike in TableGen, this can't be easily inferred, as operand types are not specified.

While CoreDSL 2 supports defining the instruction's assembly name (would be `"cv.add.b"`), we do not make use of this feature for compatibility with other tools. Instead, the assembly name is inferred from the CoreDSL 2 identifier (`CV_ADD_B`).

Behavior

The behavior parallels the Instruction Selection Patterns in LLVM, the operation performed by the instructions is described in a generic language. In this case, the language used is based on C, thus allowing for implementation at a much higher level of abstraction as compared to LLVM's Instruction Selection Patterns.

The instruction behavior of `CV_ADD_B`, as shown in Listing 2.7, begins with a check if we are attempting to write to the `x0` register. In that case, the instruction has no effect, as the `x0` register cannot be overwritten in RISC-V. If the destination register is not the zero register, we perform four additions for each of the four elements in the vector.

Registers are accessed using `X[id]`, where `X` is the register file defined elsewhere in the CoreDSL 2 architectural state description, and `id` denotes the register index as previously defined in the encoding section. In Listing 2.7, we do not access entire registers, but rather individual 8-bit vector elements within registers. For this, the subscript operator is used to access regions of the register, by specifying upper followed by lower bounds (both inclusive).

CoreDSL 2 strictly operates by the principle of "no implicit loss of precision". In the example, to avoid the possibility of implicit overflow, the sum of two 8-bit elements is represented as a 9-bit integer. We need to explicitly truncate the result of the addition back to 8 bits to avoid a type error, as assignments also do not allow implicit truncation. For this purpose, we again use the subscript operator.

3 Implementation

3.1 Generation of CoreDSL 2 Descriptions

Ideally, instruction sets and extensions would be distributed as formal descriptions in a language such as CoreDSL 2. Unfortunately, this is rather uncommon and also not the case for the CORE-V extension, which is instead defined in a classical manual with pseudo-code behavioral descriptions [9]. As such, working with the CORE-V extension in CoreDSL 2 first necessitates implementing it in CoreDSL 2.

A naive approach to this would be the manual implementation of each instruction. This however is very tedious, in addition to being very inflexible, should instruction behavior or encoding ever change. To avoid these problems, we decided to automatically generate the CoreDSL 2 instruction descriptions for CORE-V. This process is implemented in Python in the "XCoreV CoreDSL2 Generator" repository and split into two parts, encoding and behavior.

3.1.1 Encoding

Fortunately, the CORE-V manual defines instruction encoding in a simple and relatively formal table. For encoding, we parse each instruction's fields from the table, and generate an equivalent encoding description in CoreDSL 2.

The only notable difficulty encountered during this process is regarding immediate signed-ness. CORE-V uses both zero-extended and sign-extended immediate operands. The type of extension is not always listed in the table however, and sometimes only described in plain text. To solve this, the table was manually amended with a trailing `s` as a marker for all signed immediates.

3.1.2 Behavior

Unlike instruction encoding, parsing behavior from the manual is very difficult. While descriptions in pseudo-code exist, these are not particularly formal and often include descriptions in plain text for edge cases or clarification. Sometimes, it is even necessary to fall back to the Verilog implementation of the CV32E40P processor for determining instruction behavior in certain edge cases. It is therefore not feasible to convert these to correct and formal CoreDSL 2 descriptions without manual intervention. [9]

As such, instruction behavior is implemented manually. To speed up the implementation process as much as possible, we created a simple but powerful macro expansion system to aid in the generation of CoreDSL 2 for CORE-V (or similar) instructions. For example, the CoreDSL 2 for all six variations of the `CV_ADD` instruction (see Table 2.1) is generated using just the code `D[i] = (A[i] + B[i])[LSIZE-1:0]`.

`D` is short for the destination register, `A` and `B` for operands; and `i` signifies a loop over all elements — a loop header is not required. Depending on which of the six variations of the instruction is being generated, the element count is deduced to be 2 or 4, and operand `B[i]` is either element i within `rs2`, an immediate, or the lowest element within `rs2`. To avoid implicit truncation, the result of the addition is explicitly truncated to the lane size, either 16 or 8 bits.

3.2 CoreDSL To LLVM

CoreDSLToLLVM is a utility to convert instruction definitions, including behavior, from CoreDSL 2 to LLVM TableGen. For each instruction defined in CoreDSL 2, CoreDSLToLLVM generates an equivalent LLVM TableGen instruction definition, and (if behavior can be simplified sufficiently) an LLVM Instruction Selection Pattern. Thus, in essence, CoreDSLToLLVM is a compiler that compiles CoreDSL 2 down to LLVM Instruction Selection Patterns. It is implemented in C++ in the "CoreDSL to LLVM" git repository.

As shown in Figure 3.1, in the process of compiling CoreDSL 2 to LLVM Instruction Selection Patterns *in turn*, CoreDSLToLLVM also makes use of LLVM. This is possible as LLVM already is fully capable of generating SelectionDAGs. To match a subsection of the SelectionDAG to a pattern, a one-to-one correspondence between the two is required. This implies that subsections of generated DAGs can equivalently be used *as a pattern*.

As such, the only difference between CoreDSLToLLVM's backend and a regular LLVM backend is that the SelectionDAG is not used for selecting instructions (as intended), but is rather output directly as an Instruction Selection Pattern. While this is obviously not an intended use of LLVM and does require some internal modifications to LLVM, this process has major advantages as compared to the manual implementation of a CoreDSL 2 compiler:

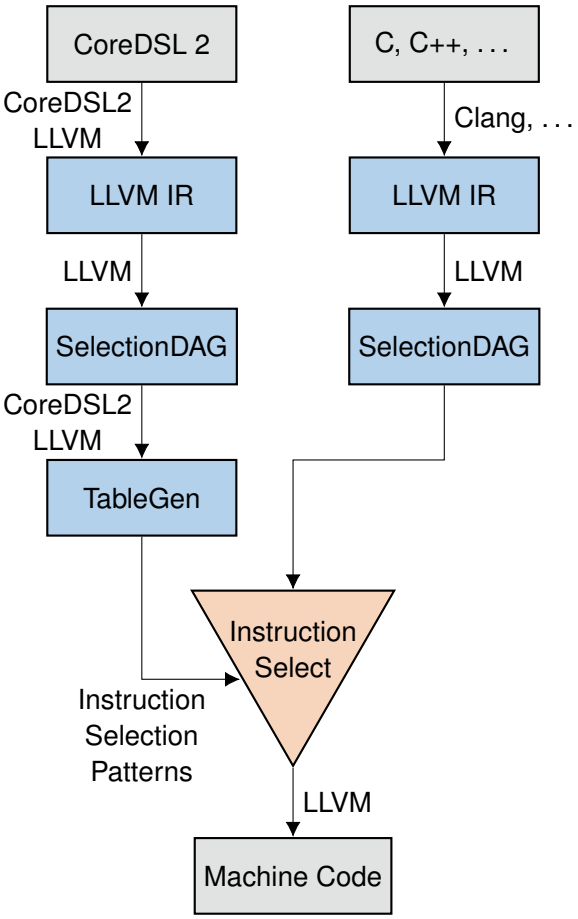


Figure 3.1 CoreDSLToLLVM Compilation Flow

- **Re-use of LLVM infrastructure**: The early steps of CoreDSL compilation, namely IR generation and optimization, are identical to those for any other programming language. As such, we can trivially use LLVM's implementations of these for CoreDSL 2, avoiding re-implementing them ourselves.
- **Natural likeness of generated patterns to DAGs of actual code**: As the generated selection patterns themselves are subject to the same LLVM compilation pipeline as any other code, the patterns will likely closely resemble the DAGs generated when compiling other code. It follows that pattern matches, and thus generation of specialized instructions are more likely than with patterns generated by other means.
- **Usage of instruction definitions for verification**: As instruction definitions are initially converted to generic LLVM IR, they can also be compiled to regular functions via LLVM's classical compilation pipeline. Said functions can then be used to implement the same instructions in RISC-V simulators. Among other things, simulators can be used for co-simulation with actual hardware description language implementations of the instructions. Both verification and compilation environments are then based on the same formal instruction definitions, minimizing the chance of bugs caused by mismatches.

3.2.1 Frontend

As with any compiler, CoreDSLToLLVM's frontend converts the textual representation of the programming language to a format more suitable for analysis and processing, such as an *abstract syntax tree* (AST) or another form of *intermediate representation* (IR). Being an LLVM-based compiler, the CoreDSLToLLVM frontend generates LLVM IR. This process is generally split into two steps, *lexing* and *parsing*.

Lexing As a first step, the stream of characters making up the source code string is converted to a stream of tokens in a process called *lexing*. Lexing strips unnecessary symbols such as comments or white space from the program and simplifies following processing steps by combining keywords and identifiers into a single symbol. For example, while an `"if"` would be represented as the characters `'i'` followed by `'f'` in a string, they are combined into a single if-token after lexing.

Parsing After lexing, the stream of tokens is *parsed*. During parsing, we interpret the sequence of tokens according to a grammar. The grammar specifies what sequences of tokens are legal, and if legal, what they symbolize. The parser then outputs some representation with which compilation is continued, ranging anywhere from graphs such as ASTs, to programming languages such as LLVM IR, or custom data structures.

While utilities and libraries exist to implement lexers and parsers, the use of these may be complicated and inflexible. As parsing C-style code is a well-studied and simple problem [20], we decided to implement lexing and parsing ourselves, thus avoiding any additional dependencies. The lexer uses a combination of a `switch` for finding simple tokens, and a string-indexed hash map for finding both static keywords and new runtime-defined identifiers. The parser is split into two sections, one for parsing instruction format, the other for instruction behavior.

Instruction Format

As previously shown in Listing 2.7, instruction definitions include both a binary `format` field as well as a textual `assembly` field. Both of these are trivial to parse. The assembly field is just one string literal; while encoding merely consists of a list of constants and variables making up the instruction word. The result of parsing both of these fields in CoreDSLToLLVM is a `CDSLInstr` struct, in which the equivalent information is easily accessible. The struct is used during pattern generation to access information about the instruction and is also printed as an LLVM TableGen instruction definition similar to Listing 2.6.

Instruction Behavior

Instruction behavior is parsed to LLVM IR by CoreDSLToLLVM. This is a significantly more involved process than the previous parsing of the instruction format, as C syntax — unlike CoreDSL instruction format syntax — allows for nested statements and expressions. However, in essence, it just entails the construction of an LLVM frontend, a program that converts an input programming language to LLVM IR. This is exactly what the LLVM C++ library was designed for, and is thus a well-documented and generally well-understood process [20].

CoreDSLToLLVM converts the token stream generated by the lexer directly to LLVM IR during parsing, skipping any intermediate steps such as the construction of an AST. This is possible as the input language is simple; and generated LLVM IR not at all needing to be optimal, as it is heavily optimized in the following steps. Thus, statements and expressions in CoreDSL 2 can be translated to equivalent LLVM IR code in a naive manner.

Behavior Functions Every instruction behavior block is parsed to an equivalent *function* in LLVM IR. Shown in Listing 3.2 is the function generated for the behavior of `cv.add.sci.h`. The function has multiple parameters, the first three of which are pointers to registers. The final arguments are values of the instruction's immediate operands, only the first of which is used in this case.

To implement the behavior of `cv.add.sci.h` in LLVM IR, we first calculate a pointer to the lowest element in `rs1`. Then, this element is loaded and extended to 32 bits.[1] After loading, the immediate operand is added, and the lower 16 bits of the result are stored in the destination register. The same process is repeated once more for the element in `rs1` with index 1.

Optimizations While, as mentioned, LLVM IR is generated rather naively by the CoreDSLToLLVM parser, we have made some adjustments to improve LLVM-generated code for use in pattern generation:

- **Use of pointers to registers**: A natural way to implement input operands and the result would be by-value function arguments and a return value, respectively. This is possible, but limits optimization potential with LLVM, as vectorization is not performed for slices of scalar in-register values. Additionally, this allows for trivial modification of specific bytes in the destination register, as memory is accessed in byte granularity.
- **`noalias` specifier for destination register pointer**: The destination pointer is declared `noalias` (equivalent to `restrict` in C) to specify that memory accessed via the pointer does not overlap with that accessed by any other pointers, in particular, the source register pointers [15]. Without this, vectorization would often require runtime checks for overlaps. Note that, when running the function in e.g. a simulator, this implies that the destination register pointer points to a temporary value, to make sure modifications to it are *not* visible in source operands while the instruction behavior function is executing.
- **Use of `extractelement` instead of shift and mask where applicable**: LLVM readily converts a cast to a vector followed by an element access to a shift and mask (if deemed more optimal). In our experience, the inverse conversion is *not* readily performed by LLVM. Thus, we always generate a cast to a vector followed by an `extractelement` if the accessed slice is aligned correctly and of correct size.
- **Use of power-of-two integer sizes**: CoreDSL 2 supports and makes heavy use of integers of arbitrary bit width. While LLVM also supports integers of any length, these are not widely used for compilation of languages like C. This may lead to patterns slightly different from DAGs generated when compiling equivalent C code. To avoid this, we round all integer sizes up to the next power of two.
- **Removal of "`if (rd != 0)`" check**: By convention, CoreDSL 2 instruction descriptions check if the destination register is the `x0`, and do not write a result if so. This is as `x0` is a constant zero register in RISC-V, which cannot be overwritten. In our case, these low-level considerations are resolved by LLVM's Register Allocator. As such, we ignore this check by compiling it to a never-taken branch, which is subsequently optimized out by LLVM.

[1] This extension is the result of CoreDSL 2's integer extension behavior — even though we only use 16 bits of the result, the temporary intermediate value has to be 17 bits, which is rounded up to 32 bits by the parser.

```
CV_ADD_SCI_H {
  encoding: 5'b00000 :: 1'b0 :: Imm6[0:0] :: Imm6[5:1] :: rs1[4:0] :: 3'b110 :: rd[4:0] :: 7'
      b1111011;
  assembly: "{name(rd)}, {name(rs1)}, {Imm6}";
  behavior: {
    if (rd != 0) {
      // (to sign-extend in CDSL2, we first cast to signed, then expand)
      X[rd][15: 0] = (X[rs1][15: 0] + ((unsigned<16>)((signed)Imm6)))[15:0];
      X[rd][31:16] = (X[rs1][31:16] + ((unsigned<16>)((signed)Imm6)))[15:0];
    }
  }
}
```

Listing 3.1 CoreDSL 2 instruction definition for cv.add.sci.h (add immediate to both 16-bit elements in rs1)

```
define void @implCV_ADD_SCI_H(
    ptr noalias %0, ptr %1, ptr %2,
    i32 %3, i32 %4) {

  // "if (rd != 0)" is compiled to this
  // branch, and later optimized out.
  br i1 true, label %6, label %25

  6:
  %7 = getelementptr i16, ptr %0, i32 0
  %8 = getelementptr i16, ptr %1, i32 0
  %9 = trunc i32 %3 to i16
  %10 = load i16, ptr %8, align 2
  %11 = zext i16 %10 to i32
  %12 = zext i16 %9 to i32
  %13 = add i32 %11, %12
  %14 = bitcast i32 %13 to <2 x i16>
  %15 = extractelement <2 x i16> %14, i32 0
  store i16 %15, ptr %7, align 2
  %16 = getelementptr i16, ptr %0, i32 1
  %17 = getelementptr i16, ptr %1, i32 1
  %18 = trunc i32 %3 to i16
  %19 = load i16, ptr %17, align 2
  %20 = zext i16 %19 to i32
  %21 = zext i16 %18 to i32
  %22 = add i32 %20, %21
  %23 = bitcast i32 %22 to <2 x i16>
  %24 = extractelement <2 x i16> %23, i32 0
  store i16 %24, ptr %16, align 2
  br label %25

  25:
  ret void
}
```

Listing 3.2 Unoptimized LLVM IR for cv.add.sci.h, as generated by CoreDSLToLLVM Parser (annotated)

```
define void @implCV_ADD_SCI_H(
    ptr noalias nocapture writeonly %0,
    ptr nocapture readonly %1,
    ptr nocapture readnone %2,
    i32 %3, i32 %4) local_unnamed_addr #0 {

  // Truncate Immediate
  %6 = trunc i32 %3 to i16

  // First Operand
  %7 = load <2 x i16>, ptr %1, align 2

  // Second Operand (replicate immediate
  // on all lanes via insert & shuffle)
  %8 = insertelement <2 x i16> poison,
    i16 %6, i64 0
  %9 = shufflevector <2 x i16> %8,
    <2 x i16> poison,
    <2 x i32> zeroinitializer

  // Compute and write result
  %10 = add <2 x i16> %7, %9
  store <2 x i16> %10, ptr %0, align 2
  ret void
}
```

Listing 3.3 LLVM IR in 3.2 after LLVM Optimization Pipeline (annotated)

3.2.2 Optimization and Vectorization using LLVM

In CoreDSL 2, instruction behavior can be arbitrarily complex, as it is defined in a relatively unrestricted C-style programming language. The same cannot be said for LLVM Instruction Selection Patterns, which are defined in a very restricted language, and thus cannot represent arbitrary computation. Therefore, only a subset of code that can be represented in CoreDSL 2, or after parsing in LLVM IR, can later be compiled into valid Instruction Selection Patterns.

The most important limitations are as follows:

- **Single basic block only**: Instruction Selection DAGs are created in isolation for all basic blocks. As such, a pattern cannot span multiple basic blocks. All loops and branches in instruction behavior definitions must to be optimized out, unrolled or otherwise linearized, such that only one basic block remains.
- **Sequence ends with exactly one write to destination register**: Instruction Selection Patterns are represented as trees, where the root of the tree represents the result value. The tree cannot have multiple or no roots. As such, patterns must have exactly one result, which is represented as a trailing store.
- **No memory access**: Instruction Patterns cannot track values written to or read from memory. Generated code therefore may not contain any temporary variables on the stack.[2]

However, the CoreDSLToLLVM parser can and will generate all of these unsupported operations. Instead of attempting to remove them itself, CoreDSLToLLVM relies on LLVM's optimizer to remove them. Fortunately, all of these operations are generally considered "slow". For example, accessing a register is usually faster than accessing memory. Thus, compiler optimizers, including LLVM's, are already targeted towards removing them.

Of course, it will always be possible to write complex instruction definitions that cannot be optimized to remove these unsupported operations. For conventionally simple instruction definitions however, such as most in the CORE-V SIMD extension, LLVM is usually capable of optimizing them to an extent sufficient for pattern generation to succeed.

Note that these restrictions only apply to the generation of Instruction Selection Patterns. When using instruction definitions in e.g. a simulator, they can be arbitrarily complex. This is because they are then compiled by LLVM's regular pipeline, which naturally supports arbitrary programs.

Auto-Vectorization

For generating SIMD instruction patterns, another important aspect is vectorization. As shown in Listings 3.1 and 3.2, both the CoreDSL 2 as well as the parser-output LLVM IR describe the SIMD instruction as multiple subsequent scalar operations. This is problematic for two reasons:

1. To later generate SIMD instructions, they must be detected as such, as their patterns need to explicitly use vector types.
2. Multiple scalar calculations produce multiple results, but a pattern may only have one result.

Fortunately, LLVM's optimization pipeline is capable of solving this problem as well, in particular, LLVM's auto vectorizer. In its intended usage, the vectorizer attempts to combine scalar operations into vector operations, thus improving performance by increasing parallelism. Just as with the general optimizations mentioned previously, this too maps neatly to compiling instruction behavior functions.

This is, as detecting whether an instruction definition defines a SIMD instruction or not is fundamentally the same problem as vectorizing scalar code. In both cases, we are attempting to describe a previously scalar computation using vector operations by detecting parallelism. For regular code, if vectorization is successful, SIMD instructions are emitted. For our instruction definitions, if vectorization is successful, the definition is changed from a scalar one to a vectorized one.

[2] This is a separate notion from accessing "RISC-V memory", e.g. for implementing RISC-V load and store instructions. Meant here is not the memory of the RISC-V processor, but rather memory that LLVM may allocate in the behavior function for e.g. register spills or temporary values on the stack. In other words, if a simulator running the optimized behavior function needs to allocate memory when running the function, we cannot create a pattern for it.

In a vectorized instruction definition, operations are performed in parallel explicitly. An example of this is shown in Listing 3.3. Instead of using two scalar `add` operations as was done in the unoptimized definition (Listing 3.2), the instruction is now described in terms of vector operations. Such as `add <2 x i16>`, which replaces the two scalar additions. This solves the previously mentioned first problem. The computation performed is now described as a vector operation, thus a pattern using vector types can be generated.

Just as important, the `load` and `store` operations, for reading from and writing to registers, have also been vectorized. With this, the instruction now produces exactly one result, namely the value that is stored, of type `<2 x i16>`. This solves the previously mentioned second problem of multiple results.

3.2.3 Backend

After LLVM IR is optimized, every basic block is converted to an Instruction Selection DAG. Then, various optimizations and legalizations are run on these DAGs. Immediately afterward, pattern matching is run on every DAG to select machine instructions covering all sections of the DAG.

In CoreDSLToLLVM, the conversion to an Instruction SelectionDAG, as well as legalization and optimization, are run as usual. However, we do not intend to select machine instructions. Instead, we interrupt the usual compilation flow, and run the CoreDSLToLLVM backend on the DAG as it exists immediately before pattern matching would usually occur. This way, exactly the same transformations are applied to the patterns we generate, as are usually applied to the SelectionDAGs patterns are matched to.

By extracting patterns as late as possible in this manner, we can achieve maximum similarity between patterns and DAGs, thereby increasing the chance of matches. Additionally, few additional transformations and little post-processing is necessary, as patterns are already in their final state. Essentially, all that remains is the conversion of LLVM's internal representation of a SelectionDAG to the textual representation of patterns used in TableGen source code, which is the output format.

Accessing the DAG

Generally, LLVM's SelectionDAG only exists temporarily and is not intended to be operated on independently of LLVM's code generator. Unlike in the generic and relatively extensible LLVM IR, it is much more difficult to access the DAG at specific points in the code generation pipeline. Usually, backend code operating on the DAG is merely called for specific transformations, such as the custom legalization of a certain operation [21].

Fortunately, one exception to this is the `PreprocessISelDAG()` function. This function allows backends to inspect and modify the entire DAG immediately before instruction selection. This happens to be the exact time at which we wish to extract a pattern from the SelectionDAG, thus making this an appropriate entry point into CoreDSLToLLVM's custom code from LLVM's regular code generation pipeline.

To override this function, we create a minimal custom LLVM Target Machine `RISCVPatternTargetMachine`, which derives from the regular LLVM `RISCVTargetMachine`. This allows us to override the necessary virtual functions to insert a custom `PreprocessISelDAG()` function. Before entering custom CoreDSLToLLVM code, the regular RISC-V `PreprocessISelDAG()` function is still run, as shown in Listing 3.4, to maintain parity between DAGs used for pattern generation and pattern matching.

Reading the DAG

The SelectionDAG is stored as a graph, with nodes of type `SDNode`. The `SDNodes` represent operations, while edges between them represent data flow or other dependencies between operations. Each `SDNode` contains references to both all its operands and all users of its result value(s). SDNodes may have zero one or multiple operands and results. In addition to the operation they perform, `SDNodes` also store the

```
void RISCVDAGToPatterns::PreprocessISelDAG()
{
  // Run regular RISC-V PreprocessISelDAG
  RISCVDAGToDAGISel::PreprocessISelDAG();
  // Run CoreDSLToLLVM backend
  PrintPattern(*CurDAG);
}
```

Listing 3.4 Custom PreprocessISelDAG function used to access the DAG immediately before instruction selection

type of each generated result. Vector types are supported as well. A visualization of the SelectionDAG generated for `cv.add.sci.h` is shown in Figure A.1.

In CoreDSLToLLVM, this LLVM-specific DAG is first converted to a custom DAG representing the same computation. The custom DAG is very similar to LLVM's representation, but minimal and easily modifiable for pattern generation. As the DAGs are similar, we generally just recurse through LLVM's DAG, while re-creating the same structures within the custom DAG.

Additionally, while creating the custom DAG, we convert various dummy representations to their concrete counterparts. For example, operand register access has previously been represented as loading a value from a register source pointer (see Section 3.2.1). This pattern is detected and converted to a register-access node specific to our custom DAG. Similar conversions are performed for immediate operand accesses, as well as the result value.

If any unsupported operations or otherwise unsupported features are encountered during conversion to a custom DAG, the conversion process will fail. As described previously, only a subset of instruction behavior descriptions can be encoded in a pattern. If construction fails, the behavioral definition, or LLVM's optimization or legalization settings, can possibly be tuned manually for successful conversion. Otherwise, this is largely expected behavior for instruction definitions that are too complex.

The conversion process is generally performed as follows:

1. Control Flow is not supported: If multiple DAGs exist for a single instruction implementation, abort.
2. Check if the last node before `return` is a 32-bit wide store to the destination register. If not, abort.
3. Recurse through `SDNodes` defining the stored value:
   - Generally, create equivalent node in custom DAG.
   - Convert load-based register access and immediate access to concrete custom DAG nodes.
   - If unsupported operations (load, store, ...) are used, abort.

Printing Instruction Selection Patterns

As necessary transformations have been performed during conversion to the custom DAG format, printing the Instruction Selection Pattern in LLVM TableGen syntax mostly entails serialization of the custom DAG format. This is achieved using simple depth-first recursion through the DAG, while printing each operand in LLVM TableGen syntax.

Finding Operand Types One minor difficulty when generating SIMD instructions is in determining register types. In the CORE-V SIMD extension, SIMD is performed in regular integer GPRs. Within LLVM however, the registers holding SIMD types must be declared separately from GPRs, as to not interfere with non-SIMD instruction patterns.

Thus, when generating instruction patterns, we must decide whether to use SIMD registers, and if so, of what type. In most cases, this is very simple. For example, if the instruction does not use SIMD at all, we exclusively use GPRs. Similarly, if only vector types are used, we use the corresponding SIMD register type for all operands. Somewhat more challenging are instructions with different operand types. For instance, these include the `cv.OP.sc` scalar replication instructions. In their case, the first operand must be a SIMD register, while the second is a GPR, as the second operand is a scalar value.

To determine the specific combination of operand types, we search for a use of each register in the pattern. If a use is found, we note the type. Not handled are instructions that use one source operand as multiple types, as these are not common in practice (none occur within the CORE-V extension). If `rd` is used as a source operand (in addition to being the destination), it may have a different type.

Apart from the pattern, we also need to specify deduced register types within the TableGen instruction definition (see Listing 2.6). As it may be required to create multiple definitions with different input and output types[3], we append the types used for each register to the instruction identifier, in the order of `rd`, `rd` (as source operand, if used), `rs1`, `rs2`. For example, the definition of `cv.add.h` is named `CV_ADD_H__V2_V2_V2` in TableGen, while `cv.add.sc.h` is `CV_ADD_SC_H__V2_V2_S`.

3.3 Using Generated Patterns in LLVM

So far, we have essentially regarded LLVM as a “black box”. Instruction behavior functions are input as LLVM IR, LLVM compiles them, and they are output as patterns. In reality, some additional adjustments to LLVM and in particular the RISC-V backend are necessary. These generally fall into two categories:

1. Patching LLVM to improve generated patterns. This prominently includes adjusting Legalizer settings as well as Heuristics, to enhance the quality of generated patterns. For example, allowing vector types in patterns.
2. Including the finalized patterns in LLVM, to be used during normal compilation. This just entails including the newly generated TableGen files, as well as some manual modifications to LLVM RISC-V infrastructure.

3.3.1 Manual Patches to LLVM

As output patterns are generated by the LLVM SelectionDAG code generator, they are subject to the same transformations as any other code. Unlike regular code, which can always rely on fallback options in case a certain operation is not supported, we are limited to what can be represented well in TableGen Instructions Selection Patterns. If LLVM fails to sufficiently simplify an instruction definition, pattern generation for said instruction will either fail outright; or generate a large, unwieldy pattern, which is unlikely to be matched when compiling code. It is thus extremely important to tune LLVM’s transformations for pattern generation.

3.3.2 Legalizer Settings

The most important manual adjustments to pattern generation are made in the Legalizer settings, as there we select which operations are allowed within patterns.

This is a balancing act: Generally, allowing additional operations within patterns will increase their conciseness. This is a very desirable trait, as small, concise patterns are much more likely to match later on, as compared to complex ones. For example, if operations on vectors are not allowed, patterns for SIMD

[3] This is currently not strictly required, as only one automatic pattern per instruction is generated, but simplifies implementation of manual patterns.

```
def : Pat<(v4i8 (splat_vector GPR:$rs2)),
  (CV_ADD_SC_B X0, GPR:$rs2)>;

def : Pat<(v2i16 (splat_vector GPR:$rs2)),
  (CV_ADD_SC_H X0, GPR:$rs2)>;
```

Listing 3.5 Manually implemented **splat_vector** patterns for v4i8 and v2i16.

instructions *can* still be generated.[4] The operation will however be expressed as multiple *scalar* operations instead of a single vector operation. This involves significantly more nodes in the pattern, all of which have to be equal to the corresponding node in the SelectionDAG, making a match much more unlikely.

On the other hand, allowing an additional operation within patterns requires an exclusive pattern for said operation. In case no other patterns match, this exclusive pattern is used as a fallback for converting the operation to machine instructions. If this fallback does not exist, match failures and thus compiler crashes will occur.

This requirement is unproblematic for operations for which precisely corresponding to machine instructions exist, as the pattern generated for the corresponding machine instruction will be an exclusive pattern for the operation. For instance, the `add` operation on a vector of type `v4i8` perfectly corresponds to the `cv.add.b` instruction. An exclusive pattern for `(v4i8, add)` will thus be generated for the instruction (see Listing 2.4).

If no matching instruction exists however, the operation has to be *emulated* using multiple or reconfigured machine instructions. Currently, these emulation patterns cannot be generated by CoreDSLToLLVM, as CoreDSLToLLVM only generates patterns for single machine instructions in their default configuration. Therefore, emulation patterns have to be implemented manually for otherwise unsupported operations.

When compiling the CORE-V SIMD extension, these e.g. include the `splat_vector` operation. It is manually implemented as shown in Listing 3.5. The first operand of a scalar-replication add is fixed to `x0`, the constant zero register. We are thus left with the scalar in `rs2` replicated on all lanes. This "operand-fixing" cannot be performed by CoreDSLToLLVM, which is why these patterns have to be implemented manually.

DAG Operation	Exclusive Pattern
`add`	*automatic*
`sub`	*automatic*
`and`	*automatic*
`or`	*automatic*
`xor`	*automatic*
`shl`	*automatic*
`sra`	*automatic*
`srl`	*automatic*
`abs`	*automatic*
`umax`	*automatic*
`umin`	*automatic*
`smax`	*automatic*
`smin`	*automatic*
`setcc`	*automatic*
`build_vector`	TableGen
`bitcast`	TableGen
`vecreduce_add`	TableGen
`splat_vector`	TableGen
`vector_extract`	TableGen
`vector_insert`	TableGen/C++
`vector_shuffle`	C++
`load`	promote to `i32`
`store`	promote to `i32`

Table 3.1 Additionally allowed operations for CORE-V SIMD, and the used method of implementing their exclusive patterns

Table 3.1 shows all operations which are enabled for vector types `v4i8` and `v2i16`, and the implementation of the respective exclusive pattern (or alternative method of generation). The majority of operations are simple arithmetic, which generally have a counterpart in the CORE-V SIMD extension. Their exclusive patterns are thus generated automatically. Exceptions are mostly more specialized operations.

Custom TableGen Patterns Of these, most can be implemented using manual patterns in TableGen, as previously demonstrated for `splat_vector` in Listing 3.5. Another example is `bitcast`, which is used by LLVM to convert between vector and integer types. In CORE-V SIMD, vectors and integers are stored in

[4] Assuming we do still allow operations to create vectors from scalars, and to store vectors, as patterns may only have one result.

the same registers, therefore this operation expands to *nothing*, it is simply removed. Edge cases like this one are conceptually simple, but difficult to recognize programmatically, and thus implemented manually.

Custom Legalize Settings `load` and `store` are easy to implement, but their implementation is not written in TableGen as an Instruction Selection Pattern. Instead, vector `load` and `store` operations are promoted to type `i32`. This again works as vectors are in the same registers as scalars. Promotion is a *Legalize Action*, which is specified in the Legalizer settings. These settings cannot be automatically generated by CoreDSLToLLVM, and therefore have to be set manually.

Custom Matching Finally, some instructions may fundamentally not fit LLVM's assumptions. Their matching to machine code is thus partly or entirely implemented manually in C++, as is intended in LLVM for operations that do not fit any other methodology [5]. In our case, `vector_insert` can only be reasonably implemented with CORE-V SIMD instructions for constant indices. As such, custom legalization code is used, which only allows `vector_insert` with a constant index. Another case is `vector_shuffle` where an immediate describing the shuffle to perform has to be generated from LLVM's internal representation.

3.3.3 Other Patches

LLVM backends, including the RISC-V backend, implement a wide variety of functions to estimate various quantities or report the availability of features. We refer to these functions as *heuristics* here. Heuristics are somewhat similar to the previously discussed Legalizer settings, though they are used more generally; as opposed to the Legalizer which mainly operates on SelectionDAG.

A number of heuristics in the RISC-V backend have to be adjusted for the successful compilation and inclusion of SIMD instructions. These include very basic heuristics, to specify that SIMD is supported at all — otherwise, the LLVM optimizer would not attempt to vectorize loops. Other heuristics include those that determine whether to unroll loops, or heuristics to estimate register usage for a basic block in LLVM IR. An exhaustive list of patches can be found in the commit log of the CORE-V LLVM Project git repository, branch `pattern-gen`.

Besides heuristics, another required manual change is the definition of new register types `PulpV4` and `PulpV2` in LLVM RISC-V TableGen infrastructure. These are defined to store vectors of type `v4i8` and `v2i16` respectively, and are used for vector input and output values in instructions definitions and patterns.

3.4 CoreDSL JIT

In addition to generating instruction selection patterns from LLVM IR instruction definitions, the CoreDSLToLLVM repository also includes the minimal instruction set simulator `cdsljit`, capable of running instructions defined in CoreDSL 2 and official `rv32im` instructions. The simulator is not fully featured, but has mainly been implemented for debugging and benchmarking purposes. Also, it serves as a demonstration of using the CoreDSLToLLVM's instruction definitions for purposes other than pattern generation. As such, we will only give a brief overview of the simulator.

3.4.1 Implementation

As the name implies, the simulator is based on just-in-time (JIT) compilation. Fragments of RISC-V code are translated to native (usually x86-64) code, and then executed. This is a common technique also implemented by instruction set simulators such as QEMU[5] and ETISS[6].

We translate sections of RISC-V code by building equivalent LLVM IR, which is then JIT-compiled using LLVM itself[7]. At a high level, the simulator operates as follows:

1. Check if code at current program counter has been translated already.
2. If not: Generate LLVM IR for code at program counter and compile/translate to native code.
3. Run translated code.
4. When translated code encounters a branch, go back to step 1 with new program counter.

If a CORE-V SIMD or otherwise CoreDSL-defined instruction is found when translating RISC-V code, we simply emit a `call` to the matching instruction implementation function, such as the one shown in Listing 3.2 for `cv.add.sci.h`. Immediate operands and pointers to *source* registers are passed directly. As mentioned previously, a pointer to a temporary value is used for the destination register. This is as the pointer is declared `noalias`, and as such may not overlap with source register pointers.

[5] https://www.qemu.org/docs/master/devel/index-tcg.html (accessed 2023-08-10)
[6] https://github.com/tum-ei-eda/etiss (accessed 2023-08-10)
[7] https://www.llvm.org/docs/MCJITDesignAndImplementation.html (accessed 2023-08-10)

4 Results

The results of this work are twofold:

1. The pattern generated themselves, including the success rate of pattern generation, as well as the quality of patterns.
2. The assembly output, as well as the performance of code and benchmarks compiled *using* the generated patterns.

In this section, we will list the results of and analyze these two items. In both cases, the versions of relevant software used are as follows:

- **CoreDSLToLLVM**: "CoreDSL to LLVM" git repository, branch "main",
  commit `08d0cc24f9171f992f8d9aa1c90d194760576754`.
- **LLVM**: "CORE-V LLVM Project" git repository, branch "pattern-gen",
  commit `ac64e908362161b85159a44886600a523270c69b`.
- **CORE-V CoreDSL2 Generator**: "XCoreV CoreDSL2 Generator" git repository, branch "main",
  commit `f49e6bc6982207a347f367a81550cf6d8d74bc88`.
- **LLVM-GEN**: "llvm-gen" git repository, branch "main"
  commit `cd2970491a87ed739835c9136d58ce86eff8b575`

4.1 Generated Patterns

The CORE-V SIMD extension defines a total of 220 instructions. All of these are implemented in the CoreDSL 2 source files generated by the CORE-V CoreDSL2 Generator, namely in `XCoreVSIMD.core_desc` in the git repository. Pattern generation succeeds for 190 of these, or about 86%. Using an optimized build of LLVM and CoreDSLToLLVM, the total compilation time from CoreDSL 2 to TableGen is around 200ms, running on an Intel Core i9-9900k processor.

Vector, Vector	Vector, Scalar	Vector, Immediate Scalar
`(add PulpV4:$rs2, PulpV4:$rs1)`	`(add PulpV4:$rs1, (v4i8 (splat_vector GPR:$rs2)))`	`(add PulpV4:$rs1, (v4i8 (splat_vector (i32 simm6:$imm))))`
`(add PulpV2:$rs2, PulpV2:$rs1)`	`(add PulpV2:$rs1, (v2i16 (splat_vector GPR:$rs2)))`	`(add PulpV2:$rs1, (v2i16 (splat_vector (i32 simm6:$imm))))`

Table 4.1 Patterns generated by CoreDSLToLLVM for the six variations of simple CORE-V SIMD ops, here `cv.add`. Upper row: 8-bit elements. Lower row: 16-bit elements.

4.1.1 Basic Patterns

The majority, or 120 instructions, are compiled to patterns similar to the ones shown in Table 4.1, with `add` replaced by the particular operation. Note that only the patterns themselves are shown, not the whole TableGen definition statement as previously. Out of the six variations shown in the Table, the specific one generated depends on the type as well as the instruction variant, see Table 2.1.

Pattern generation is very simple for these instructions, as a directly matching SelectionDAG operation exists for all of them. LLVM is able to simplify the instruction definitions such that only this particular operation remains. In addition, a `splat_vector` is generated for scalar replication variants. Note that these operations are the same ones mentioned earlier in Table 3.1 as having automatically generated exclusive patterns. In particular, these are the "Vector, Vector" patterns from Table 4.1.

4.1.2 Complex Patterns

The remaining 70 instructions for which pattern generation succeeds *do not* have a direct counterpart in SelectionDAG. As such, their patterns are more complex. As mentioned previously, complex patterns are less desirable than simple ones, as they are less likely to match. However, though complex, many of these patterns will still match at times, allowing for the generation of the associated instruction. They are thus much preferred to having no pattern at all.

Complex patterns are one of CoreDSLToLLVM's biggest strengths. Simple patterns, like the ones shown before, can be written by hand with relative ease. One can also attempt to write complex patterns by hand, it is however unlikely that these express the computation in the same way that LLVM does. With CoreDSLToLLVM, patterns are compiled by LLVM itself, and will thus have a natural similarity to the DAGs of other code compiled with LLVM.

Dot Products

Dot Products are among the most powerful instructions in the CORE-V SIMD extension. Unfortunately, they also are among the instructions matching LLVM's concepts least, thus complicating the creation of their patterns.

For one, SelectionDAG does not have a dot product operation. Operations do exist for multiplying, as well as for computing the horizontal sum of a vector (`vecreduce_add`). As is usual in LLVM however, these are *truncating* operations. As such, an 8-bit dot product implemented using them would output only an 8-bit result. The CORE-V SIMD dot products are capable of 32-bit results though, avoiding the possibility of integer overflow for 8-bit vectors. The default way of achieving non-truncating operations in LLVM is by extending the types beforehand. Unfortunately, this is impossible here, as the legalized DAG may only have vector types `v4i8` and `v2i16`.

Given this, the only way to implement a non-truncating SelectionDAG pattern for CORE-V SIMD dot products is by using a *scalar* pattern. An example of this is shown in Listing 4.1. In the pattern, all calculations are performed in the scalar `i32` type. As such, the pattern is quite verbose, as compared to previously shown vectorized patterns. However, if this pattern matches, all operations can be replaced by single `cv.sdotup.b` instruction (unless the intermediate values are also used elsewhere).

```
(add
  (add
    (add
      (mul
        (and (vector_extract PulpV4:$rs2, (i32 0)), (i32 255)),
        (and (vector_extract PulpV4:$rs1, (i32 0)), (i32 255))),
      (mul
        (and (vector_extract PulpV4:$rs2, (i32 2)), (i32 255)),
        (and (vector_extract PulpV4:$rs1, (i32 2)), (i32 255)))),
    (add
      (mul
        (and (vector_extract PulpV4:$rs2, (i32 1)), (i32 255)),
        (and (vector_extract PulpV4:$rs1, (i32 1)), (i32 255))),
      (mul
        (and (vector_extract PulpV4:$rs2, (i32 3)), (i32 255)),
        (and (vector_extract PulpV4:$rs1, (i32 3)), (i32 255))))),
  GPR:$rd)
```

Listing 4.1 Pattern generated by CoreDSLToLLVM for `cv.sdotup.b`

4.1.3 Failing Instructions

Pattern generation fails for 30 of the 220 CORE-V SIMD instructions. This makes for a failure rate of about 14%. Disregarding instructions with trivially generated patterns, as described in Section 4.1.1, the failure rate rises to 30%.

Pattern generation mainly fails due to two factors for these 30 CORE-V SIMD instructions:

1. **Writing to only part of the destination register**: For example, `cv.cplxconj` negates only the upper 16-bit element within a register (complex conjugate). CoreDSLToLLVM currently requires that the entire destination register is written to.
2. **Accessing a vector element using a non-constant index**: For example, `cv.insert` and `cv.extract` allowing accessing individual elements within a SIMD register. The index of the element accessed is non-constant.[1] This is currently unsupported in CoreDSLToLLVM.

As will be discussed later, both of these are limitations within CoreDSLToLLVM and can likely be resolved with further development.

4.2 Benchmarks

In the preceding section, we analyzed the patterns generated by CoreDSLToLLVM. Of course, the goal of generating patterns is to use them in the compilation of other, generic code. Therefore, in this section, we analyze the performance of code compiled with a version of LLVM that includes the previously discussed automatically generated CORE-V SIMD patterns. As a baseline, we use the same version of LLVM, but with CORE-V SIMD patterns *disabled*.

The benchmarks we run are in two broad categories, namely those that are purely synthetic, and those that intend to emulate real-world code.

[1] From the perspective of an Instruction Selection Pattern, immediate value indices are also not constant. This is as they may change between instances of an instruction.

4.2.1 Synthetic Benchmarks

The synthetic benchmarks we use are simple functions performing common *linear algebra* or *string handling* computations. Our intention is to analyze the usefulness of generated patterns *independently* of LLVM's vectorization performance. As such, all of these functions are written in such a way as to be easily vectorizable for LLVM's optimizer. The final performance therefore largely depends on our custom backend and patterns, and their ability to represent LLVM's vectorized code efficiently.

The specific functions used are shown in Listing A.1. To simplify auto-vectorization, destination array pointers are as marked `restrict`. Also, the `to_upper` function for strings is not implemented using *null termination*, as would be usual in C, but uses an explicit length parameter, as the other functions do.

Environment

Measurements are performed using the minimal instruction set simulator `cdsljit`, which is included in the CoreDSLToLLVM repository. In all cases, we measure dynamic instruction count, i.e. how many instructions were executed during the program's runtime. For more meaningful results, the total is split into counts for each instruction type. Note that dynamic instruction count *does not* take into account possible hardware-implementation-specific factors such as pipeline stalls and flushes, unaligned memory penalties, or branch prediction.

For all functions except `to_upper`, the input arrays are randomized with a set seed by `cdsljit`. For `to_upper`, we compare different ASCII text inputs. In all cases, the value used for the length parameter `n` of all functions is given in the figure's title.

In addition to benchmarking, `cdsljit` dumps the arrays after execution ends. As a basic test for correctness, these dumps are compared for SIMD and non-SIMD binaries. There are no mismatches for the benchmarks presented here.

The specific testing environment used can be found in the `testing` subdirectory of the CoreDSLToLLVM git repository. The Clang and LLVM flags relevant for performance are as follows:

- `-O3`: aggressive optimization
- `-Xclang -target-feature -Xclang +unaligned-scalar-mem`: allow unaligned memory access
- `-Xclang -target-feature -Xclang +no-default-unroll`: use custom (aggressive) loop unrolling
- `-mllvm --prefer-inloop-reductions`: reduce to scalar in every loop iteration (for dot-products)
- `-mllvm --lsr-term-fold`: optimize loop iterator variables for RISC-V `bne` or `beq` instructions

In our custom version of LLVM, many of these optimizations are enabled by default. This is not the case in mainline (official) LLVM however. As such, all equivalent flags are listed here for the sake of completeness.

Vector Operations on Arrays

Figure 4.1 shows dynamic instruction counts of variations of the `add` and `dot` functions, all run with arrays of 65536 elements. The SIMD implementations were successfully compiled by LLVM to use the ideal CORE-V SIMD instructions. In particular, `cv.add.(b|h)` for add, and `cv.sdotsp.(b|h)` (accumulating dot product) for dot. The speedups achieved by the SIMD implementations are 2.14x, 4.23x, 2.77x and 4.96x respectively, in the same order as shown in the chart.

Vectorization Observe that dynamic instruction counts for the non-SIMD versions of the functions do not change much between the 8 and 16-bit versions. This is because, in scalar code, all operations are actually computed in the native 32-bit integer type and then truncated to the desired result size. As such, the most significant change from 8 to 16-bit is the usage of `lh` and `sh` instructions (load/store halfword) instead of `lb` and `sb` (load/store byte).

This is not the case for SIMD variants of the functions. There, 2 or 4 values are *packed* into a single 32-bit register and then operated on with the respective SIMD instruction. Also, the memory access instructions used are `lw` and `sw` (load/store word), which load/store 2 or 4 values at once for 16 or 8-bit variants respectively. As such, for SIMD variants, 8-bit functions are roughly twice as fast as 16-bit ones, with a constant element count.

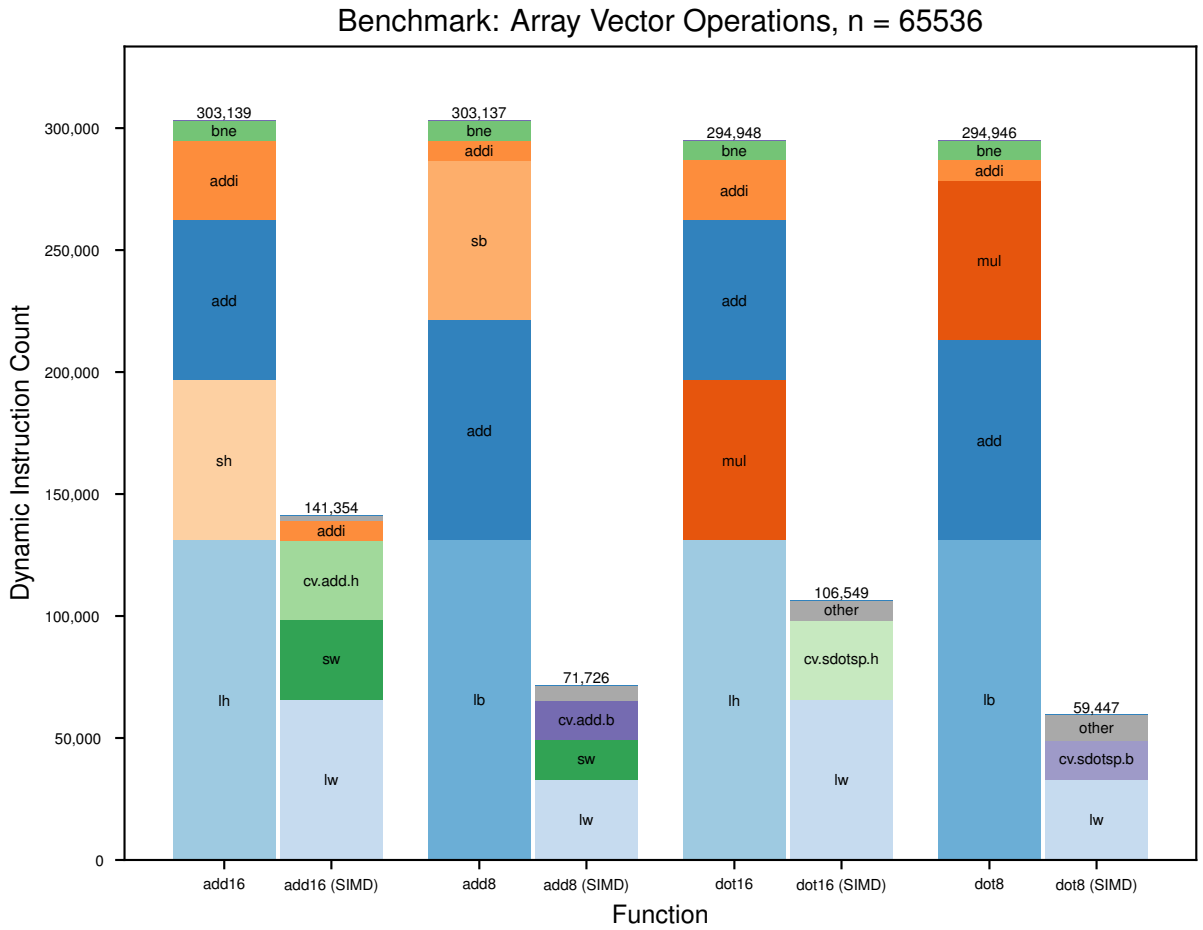


Figure 4.1 Dynamic instructions counts (separated by instruction) for 16 and 8-bit versions of add and dot, with and without CORE-V SIMD.

Matrix Multiplication

Another synthetic test function is `matmul`, which performs matrix multiplication. Note that, when using CORE-V SIMD, if possible, matrix multiplication should be performed as AB^{T} for matrices A and B, i.e. multiplying one matrix with the transpose of another. In that case, the problem simply decomposes into running the `dot` function (see A.1) to compute every element in the resulting matrix. As such, the same speedup as shown in Figure 4.1 for `dot` will be achieved trivially.

In this case, we analyze the usual AB formulation of matrix multiplication, as performance for `dot` has already been evaluated. For ease of vectorization, we use a SIMD-optimized implementation of matrix multiplication with only sequential memory accesses, as shown in Listing A.1.

Extended CORE-V SIMD Extension In addition to the baseline test with no custom instructions and the test using CORE-V SIMD, we also test matrix multiplication with an *extended* version of the CORE-V SIMD extension. In particular, this extended version adds `cv.mac.(b|h)` multiply-accumulate SIMD instructions. These instructions are *not* part of the official CORE-V SIMD extension, but are merely supposed to serve as an example and additional benchmark.[2] Just as with official CORE-V SIMD instructions, support for the new instructions has been generated using CoreDSLToLLVM. The CoreDSL 2 definitions for `cv.mac` instructions can be found in Listing A.2. Results using the extended CORE-V SIMD extension are marked as “SIMD, MAC”.

Results The results of the matrix multiply benchmarks are shown in Figure 4.2. Shown from left to right are the 16-bit versions of `matmul` without extensions, then with CORE-V SIMD, then with our extended CORE-V SIMD; followed by 8-bit versions in the same order. When moving from scalar to unmodified CORE-V SIMD code, speedups of 1.13x and 1.48x can be observed for 16 and 8-bit respectively. Comparing scalar to CORE-V SIMD including multiply-accumulate, the speedups increase to 2.48x and 4.87x respectively.

As unmodified CORE-V SIMD does not include any kind of lane-wise vector-multiply instruction, multiplication has to be performed using *scalar* RISC-V `mul` within the CORE-V SIMD implementations. Therefore, the speedup gained is not very high, as only memory accesses and addition can be expressed as SIMD instructions. Furthermore, elements have to extracted (`cv.extract`) from vectors before performing multiplication, and inserted (`cv.pack`) back into vectors after, which further decreases performance.

With CORE-V SIMD and our custom `cv.mac` instructions, we achieve much greater speedup. This is to be expected, as `cv.mac` has been implemented for this purpose, and performs precisely the operation required. Both addition and multiplication are computed in one instruction without decomposing the vector. As such, the `cv.mac` instructions demonstrate CoreDSLToLLVM’s ability to make effective use of newly defined instructions.

[2] It is likely that dot product instructions were implemented in favor of lane-wise multiply instructions. Dot products return a single scalar, and are thus less affected by the issue of integer overflow, which easily occurs with lane-wise multiply. However, due to its similarity to dot products, `cv.mac.(b|h)` (and similar instructions) can likely be implemented with minimal hardware cost, and significantly speed up computations unaffected by overflow. To alleviate the issue of overflow, one may also consider implementing instructions similar to `mulh`, returning the upper (usually truncated) half of the result; or instructions that e.g. multiply just half of the elements in `v4i8` vectors but extend to `v2i16`.

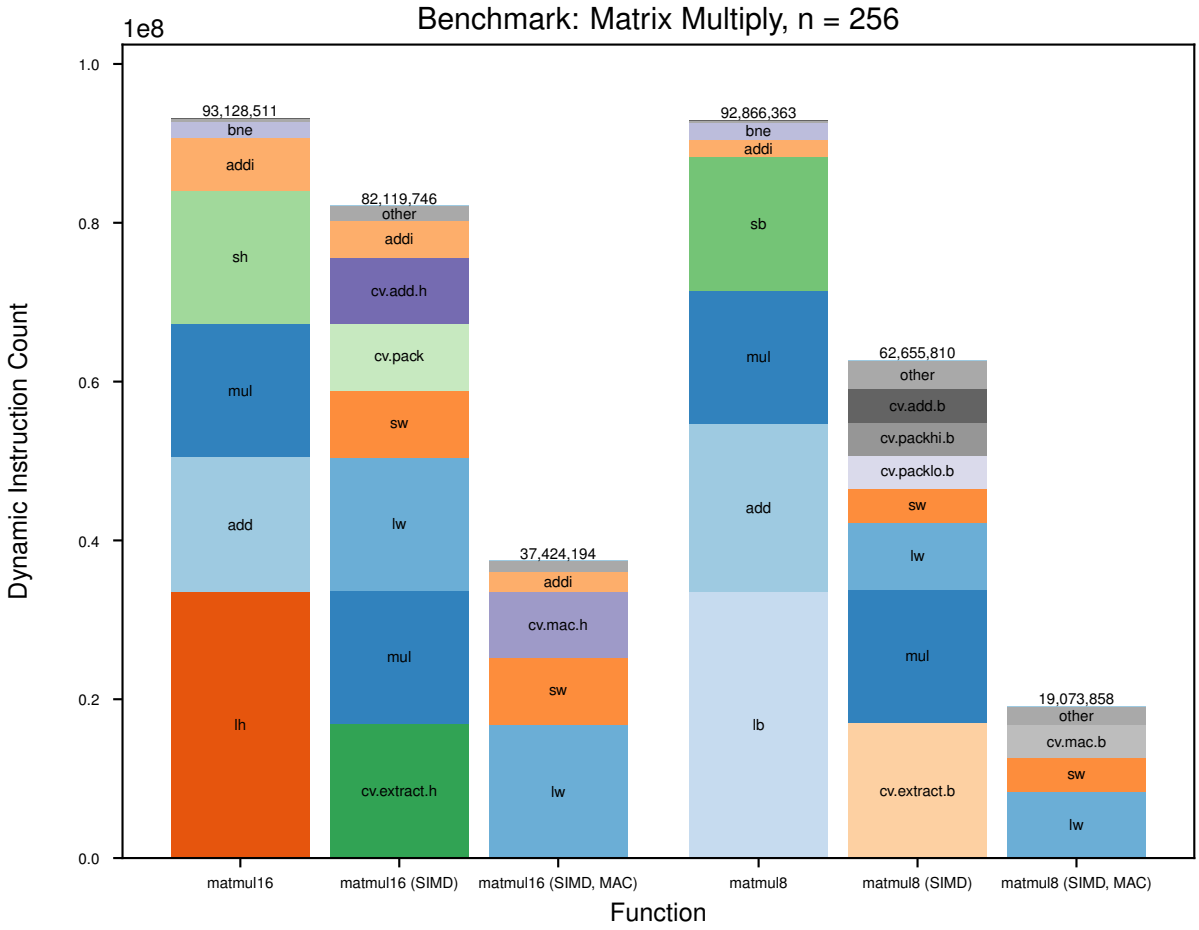


Figure 4.2 Dynamic instructions counts (separated by instruction) for 16 and 8-bit versions of `matmul`

String Operations

In all previous synthetic benchmarks, we have tested compilation and performance of linear algebra functions. Another class of problems that can benefit significantly from SIMD are operations on strings. This is because, for regular C-style strings, characters are commonly one byte in size. As such, four of them can be processed in parallel using CORE-V's 32-bit SIMD.

The example analyzed here is the `to_upper` function as shown in Listing A.1. This represents an interesting example, as branching may be involved depending on the type of character encountered. As such, we test for two inputs of only uppercase and only lowercase characters, which represent the two extremes of runtime.

The results of benchmarking are shown in Figure 4.3. The scalar version of `to_upper` uses a branch to distinguish between upper and lowercase characters. As such, a significant difference in instruction count can be observed between lowercase and uppercase input strings. The same is not true for the SIMD version, where the instruction count for both inputs is identical. The speedup achieved is 4.00x and 3.31x for lowercase and uppercase inputs respectively.

This is achieved by using SIMD masking instead of branching for the upper/lowercase case distinction, as shown in Listing 4.2.[3] As such, there is no difference in runtime depending on the input data. This example establishes that even advanced techniques like SIMD masking are used by LLVM given our automatically generated instruction patterns. In addition, the generated assembly shown in the listing demonstrates the effective use of scalar replication and immediate variants of CORE-V SIMD instructions.

[3]Note that, to keep the Listing short, loop unrolling has been disabled. The code is thus not directly comparable to that used in Figure 4.3

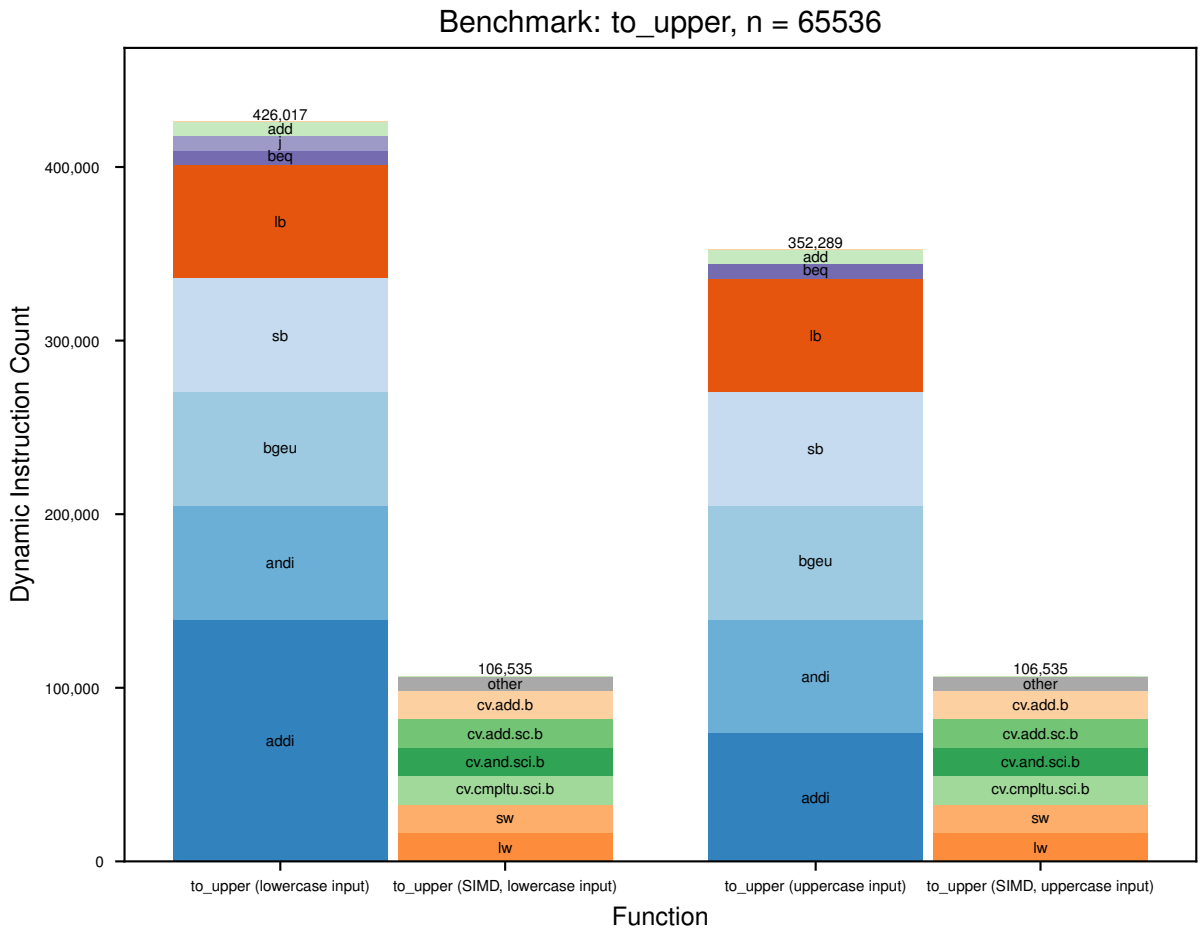


Figure 4.3 Dynamic instructions counts (separated by instruction) for `to_upper`, with lowercase-only and uppercase-only string inputs

```
// ...
li a5, -97                  // Load -'a' (ASCII) into a5
.LBB0_4:
lw a6, 0(a3)                // Load 4 characters from string
cv.add.sc.b a7, a6, a5      // Subtract 'a' from all. Values 0-25 are lowercase characters
cv.cmpltu.sci.b a7, a7, 26  // If element < 26, replace with 0xFF (otherwise 0)
cv.and.sci.b a7, a7, -32    // Replace 0xFF elements with 'A'-'a' (no change to 0 elements)
cv.add.b a6, a7, a6         // Add either zero or 'A'-'a' to original characters
sw a6, 0(a3)                // Store result
addi a3, a3, 4              // Increment pointer
bne a3, a4, .LBB0_4         // Loop back to .LBB0_4
// ...
```

Listing 4.2 Annotated, LLVM-generated assembly for `to_upper` using SIMD-masking-based predication

4.2.2 Embench

All previously shown benchmarks have been specialized simple functions ideal for vectorization, with accordingly good results. Now, we will investigate performance in benchmarks that intend to emulate the complexity of real-world tasks. In particular, we run Embench[4] with and without generated CORE-V SIMD extension support. For compilation and simulation, we use the environment provided in the "llvm-gen" git repository, using the CORE-V OVPsim instruction set simulator[5] and optimization level 3.

The dynamic instruction count reductions with CORE-V SIMD enabled are shown in Figure 4.4. For the vast majority of benchmarks, there is no or negligible change in performance. As we exclusively implement SIMD instructions, this is to be expected for tasks that LLVM fails to vectorize. For example, `matmult-int` performs matrix multiplication, but uses 32-bit integers. It is thus impossible to vectorize, given that the CORE-V SIMD vector size is 32 bits.

Some outliers exist however, most prominently `edn`, which achieves a speedup 41.3%. `edn` performs signal processing tasks, predominately using 16-bit integers, and can be vectorized. Smaller outliers are `md5sum` at 8.6% and `picojpeg` at 1.5%. In these cases, likely only small sections of code were vectorized, leading to significantly smaller speedups as compared to the previous benchmarks. A very minor negative outlier exists in `qrduino`, which creates QR codes. Likely, a minor vectorization was performed, but had a negative performance impact, leading to the slight regression in performance.

In general, performance in Embench is largely unaffected by the CORE-V SIMD extension. Outliers exist however, most of which are positive, some of which significantly so. These outliers likely represent tasks where at least part of their code uses 8 or 16-bit integers and is easily vectorizable. This result is expected, as we have implemented SIMD for 8 or 16-bit elements, and rely on LLVM's auto-vectorizer for vectorization.

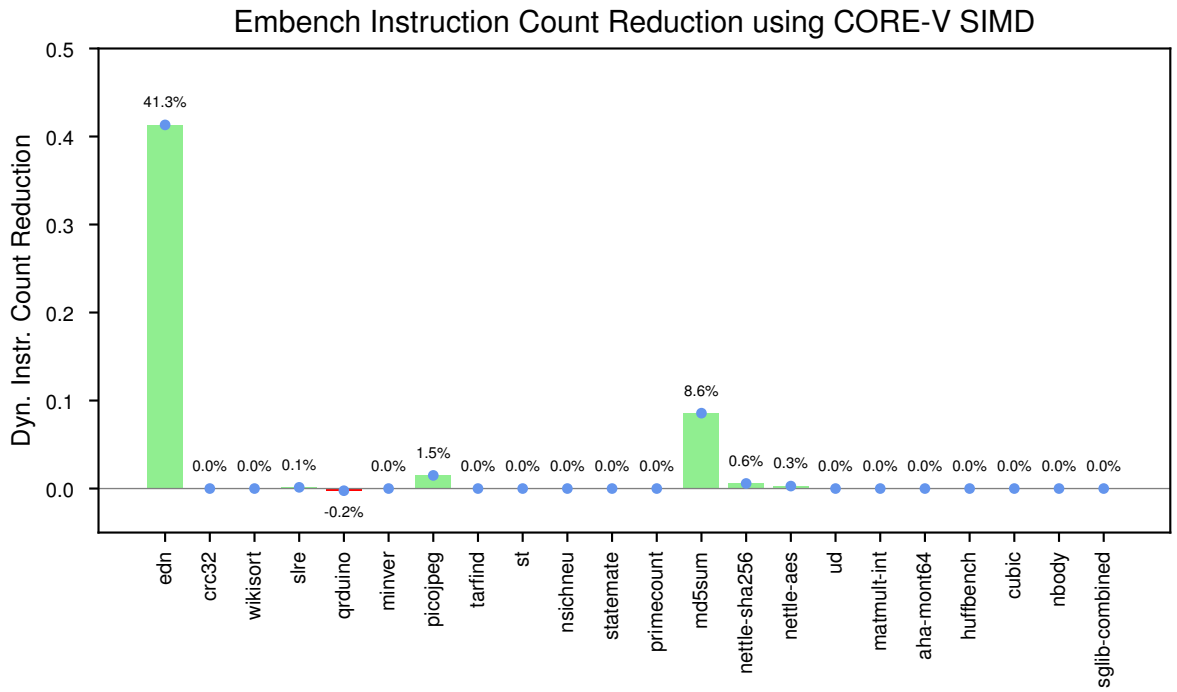


Figure 4.4 Dynamic instruction count reduction in Embench using CORE-V SIMD, relative to CORE-V SIMD disabled

[4] https://www.embench.org/ (accessed 2023-08-11)
[5] https://github.com/openhwgroup/riscv-ovpsim-CORE-V (accessed 2023-10-12)

5 Conclusion

We have implemented CoreDSLToLLVM, an LLVM-based compiler for automatic generation of LLVM Instruction Selection Patterns and Instruction Format Information from CoreDSL 2 source code. Using CoreDSLToLLVM, we were able to generate LLVM support for the vast majority of instructions in the CORE-V SIMD RISC-V extension. Using a version of LLVM that includes this generated support, we were able to achieve significant performance improvements in benchmarks using 8 or 16 bit integers in easily vectorizable workloads. As only SIMD instructions were implemented, performance in other benchmarks is largely unaffected.

We consider CoreDSLToLLVM to significantly simplify the integration of custom instructions within LLVM. While some initial manual intervention is still, and will likely always be required; once this foundation has been built, it is trivial to implement code generation support for many kinds of instructions with minimal effort. We consider this to be especially useful during the development of new extensions, as new instructions can be prototyped rapidly.

In this context, one may even consider some of CoreDSLToLLVM's limitations to be useful indicators, as these limitations generally mirror those of LLVM. When intending to spend significant resources on compiler development, it will likely be possible to work around LLVM's limitations and create adequate code generation for any instructions. If this is not the case however, one may benefit from learning early on whether an instruction can be easily implemented in LLVM, to adjust extension design accordingly.

5.1 Limitations and Future Work

In this section, we will list current limitations within CoreDSLToLLVM, and possible approaches to mitigate them. Most of these limitations are artifacts of the current implementation of CoreDSLToLLVM; though some fundamental to the chosen approach exist as well.

5.1.1 Legalizer Settings

One of the main caveats of CoreDSLToLLVM's pattern generation is in the need for manual tuning of LLVM's Legalizer Settings. These settings fundamentally affect the success and quality of generated patterns, but are currently not automatically adjusted in any way by CoreDSLToLLVM. In fact, CoreDSLToLLVM is entirely subordinate to them, as its backend runs *after* legalization.

Given the preceding statement, an obvious idea to alleviate the issue would be running CoreDSLToLLVM code *before* legalization. This does not make sense for pattern generation itself, as patterns have to match post-legalize DAGs. However, CoreDSLToLLVM could analyze pre-legalize DAGs to e.g. determine whether exclusive patterns exist for given types and operations. If an exclusive pattern exists, the operation can be allowed in the post-legalize DAGs without concern.

This process would allow automatic deduction of Legalizer settings for operations with automatic exclusive patterns, as previously shown in the upper part of Table 3.1. Others would still require manual intervention.

5.1.2 Exclusive Patterns

An issue adjacent to Legalizer Settings is that of exclusive patterns. If we want to allow a certain operation in the Legalizer to improve pattern generation, but an instruction *exactly* equivalent to this operation does not exist, an exclusive pattern has to be implemented manually. This corresponds to operations in the lower part of Table 3.1.

It is very likely that *some* operations, which do not conform to LLVM's assumptions, will always have to be implemented manually. There however may be workable approaches to generate at least a subset of these patterns automatically as well. Currently, CoreDSLToLLVM only generates patterns corresponding to *exactly one* machine instruction. It is entirely feasible to generate patterns for compositions of multiple instructions, however. Alternatively, one could restrict instruction operands, such as by fixing input registers to `x0`, to discover possibly useful edge-case behavior.

Of course, the challenge of this approach is the enormous search space to be covered — both in terms of pattern generation time, and a practical limit to the number of patterns that can be included in a build of LLVM. If promising candidate compositions are found or manually specified however, CoreDSLToLLVM could be adjusted to generate the respective patterns. This would also serve to verify assumed behavior, as LLVM will generate the expected operation only if the performed computation matches. As such, candidates do not have to be rigorously checked for correctness beforehand.

5.1.3 Other Limitations of the Implementation

As CoreDSLToLLVM has been written with the goal of generating patterns for CORE-V SIMD, some features are limited to what was required at the time for CORE-V SIMD. In general, none of these limitations are fundamental. They could likely all be resolved with continued development of CoreDSLToLLVM.

We do not support any instructions with side effects, such as branches or memory accesses. Instructions have to be 32-bit, with no more than 3 register operands and 2 immediates. The only accessible registers are 32-bit GPRs, as we only support 32-bit RISC-V. Floating-point or dedicated vector registers defined in other RISC-V extensions cannot be accessed either.

The CoreDSLToLLVM frontend only implements a subset of CoreDSL 2, excluding e.g. architectural state, helper functions, or user-defined types. A complete implementation of CoreDSL 2 is certainly possible. Whereas we just define extensions now, complete support would allow for defining an entire processor, including *all* instructions. While patterns already exist for official RISC-V instructions and thus need not be generated, this may be useful for purposes of simulation.

5.1.4 Limitations of the Approach

Fundamental limitations in CoreDSLToLLVM are generally due to its heavy reliance on LLVM, as well as the generation of LLVM TableGen patterns for instruction selection. Most notably, TableGen patterns only ever have one result value, as they are represented as trees. This issue can likely only be resolved by means of creating an entirely custom, C++-based matching system for instructions with multiple results.

Secondarily, the chosen approach relies on LLVM's SelectionDAG code generation framework. A new code generation framework, *GlobalISel*, is in development and intends to replace SelectionDAG in the future[1]. GlobalISel uses the same TableGen Instruction Selection Patterns but works in a fundamentally different manner. While GlobalISel is not yet fully usable for RISC-V, if it does become available, it may be desirable to re-implement the CoreDSLToLLVM backend using GlobalISel, to create patterns that better match it.

[1] https://www.llvm.org/docs/GlobalISel/index.html (accessed 2023-08-14)

A Appendix

```c
#include <stdint.h>
#include <stddef.h>

void to_upper(size_t n, char c[n])
{
  for (size_t i = 0; i < n; i++)
    c[i] += (c[i] >= 'a' && c[i] <= 'z') ? ('A'-'a') : 0;
}

void add8(size_t n, int8_t d[restrict n], int8_t a[n], int8_t b[n])
{
  for (size_t i = 0; i < n; i++)
    d[i] = a[i] + b[i];
}

int32_t dot8(size_t n, int8_t a[n], int8_t b[n])
{
  int32_t acc = 0;
  for (size_t i = 0; i < n; i++)
    acc += a[i] * b[i];

  return acc;
}

// This function assumes matrix d is initialized to zero
void matmul8(size_t n, int8_t d[restrict n*n], int8_t a[n*n], int8_t b[n*n])
{
  // The inner two loops are swapped, as compared to the
  // common algorithm, for sequential memory access
  for (size_t a_y = 0; a_y < n; a_y++)
  {
    for (size_t b_y = 0; b_y < n; b_y++)
    {
      for (size_t i = 0; i < n; i++)
        d[a_y * n + i] += a[a_y * n + b_y] * b[b_y * n + i];
    }
  }
}

// For each of the functions ending with "8", an equivalent 16-bit version
// exists, with no changes other than replacing all "int8_t" with "int16_t"
```

Listing A.1 Full listing of synthetic benchmark functions

```
CV_MAC_H {
  encoding: 5'b1111 :: 1'b0 :: 1'b0 :: rs2[4:0] :: rs1[4:0] :: 3'b0 :: rd[4:0] :: 7'b1111011;
  assembly: "{name(rd)}, {name(rs1)}, {name(rs2)}";
  behavior: {
    if (rd != 0) {
      X[rd][15: 0] += (X[rs1][15: 0] * X[rs2][15: 0])[15:0];
      X[rd][31:16] += (X[rs1][31:16] * X[rs2][31:16])[15:0];
    }
  }
}
CV_MAC_B {
  encoding: 5'b1111 :: 1'b0 :: 1'b0 :: rs2[4:0] :: rs1[4:0] :: 3'b1 :: rd[4:0] :: 7'b1111011;
  assembly: "{name(rd)}, {name(rs1)}, {name(rs2)}";
  behavior: {
    if (rd != 0) {
      X[rd][ 7: 0] += (X[rs1][ 7: 0] * X[rs2][ 7: 0])[7:0];
      X[rd][15: 8] += (X[rs1][15: 8] * X[rs2][15: 8])[7:0];
      X[rd][23:16] += (X[rs1][23:16] * X[rs2][23:16])[7:0];
      X[rd][31:24] += (X[rs1][31:24] * X[rs2][31:24])[7:0];
    }
  }
}
```

Listing A.2 CoreDSL 2 descriptions of multiply-accumulate instructions added to CORE-V SIMD

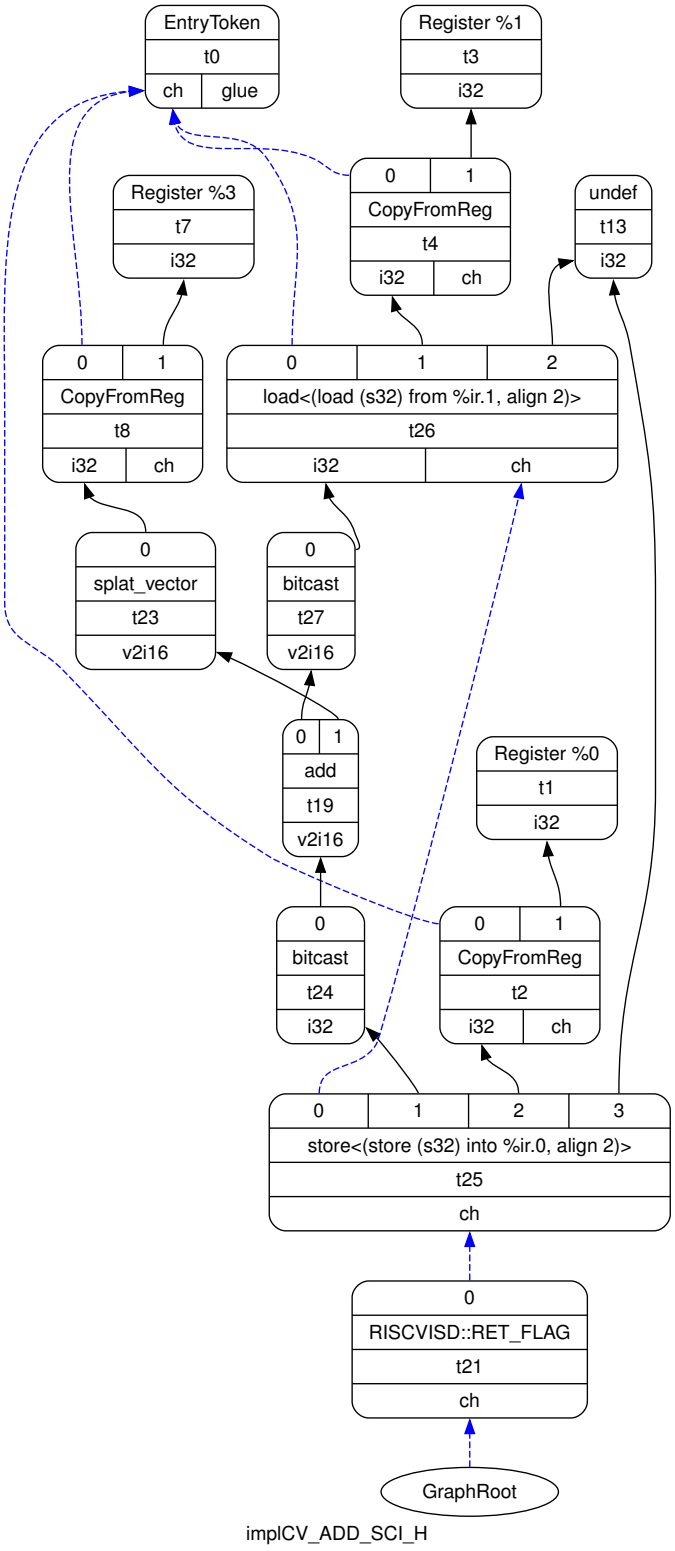


Figure A.1 SelectionDAG for `cv.add.sci.h`, generated from optimized LLVM IR in Listing 3.3. As not yet converted to custom DAG, register accesses are represented as load/store.

Bibliography

[1] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, 2019. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[2] Samuel Greengard. Will RISC-V Revolutionize Computing? *Commun. ACM*, 63(5):30–32, April 2020.

[3] Enfang Cui, Tianzheng Li, and Qian Wei. RISC-V Instruction Set Architecture Extensions: A Survey. *IEEE Access*, 11:24696–24711, 2023.

[4] LLVM User Guide for RISC-V Target, 2023. https://llvm.org/docs/RISCVUsage.html.

[5] The LLVM Target-Independent Code Generator, 2023. https://www.llvm.org/docs/CodeGenerator.html.

[6] 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*, pages 278–285, 2012.

[7] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. *Synthesizing Instruction Selection*. Karlsruher Institut für Technologie (KIT), 2017.

[8] T-Head Semiconductor Co., Ltd., VRULL GmbH. T-Head ISA extension specification, 2022. https://github.com/T-head-Semi/thead-extension-spec/releases/download/2.2.2/xthead-2023-01-30-2.2.2.pdf.

[9] OpenHW Group. CORE-V Instruction Set Custom Extensions, 2023. https://github.com/openhwgroup/cv32e40p/blob/master/docs/source/instruction_set_extensions.rst.

[10] Arm Limited. Introduction to SVE, 2023. https://documentation-service.arm.com/static/62d806a1b334256d9ea8fc37.

[11] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, 2016. https://intel.com/sdm.

[12] RISC-V Foundation. RISC-V "V" Vector Extension, 2021. https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[13] RISC-V Foundation. RISC-V "P" Extension Proposal, 2021. https://github.com/riscv/riscv-p-spec/blob/master/P-ext-proposal.pdf.

[14] The LLVM Compiler Infrastructure, 2023. https://llvm.org/.

[15] LLVM Language Reference Manual, 2023. https://llvm.org/docs/LangRef.html.

[16] LLVM's Analysis and Transform Passes, 2023. https://www.llvm.org/docs/Passes.html.

[17] LLVM TableGen Overview, 2023. https://llvm.org/docs/TableGen/.

[18] Keith Cooper and Linda Torczon. *Engineering a Compiler*. 2004.

[19] Julian Oppermann. CoreDSL 2 programmer's manual, 2023. https://github.com/Minres/CoreDSL/wiki/CoreDSL-2-programmer's-manual.

[20] My First Language Frontend with LLVM Tutorial, 2023. https://www.llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html.

[21] Writing an LLVM Backend, 2023. https://www.llvm.org/docs/WritingAnLLVMBackend.html.