LLVM Backend Support for Data Streaming Extensions

Tiago Cardoso Pires

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Filipe Simões Santos Moraes da Silva Neves
Prof. Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Nuno Filipe Simões Santos Moraes da Silva Neves
Member of the Committee: Prof. João Carlos Viegas Martins Bispo

November 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my family for always supporting me through this journey, who was always by my side every time I needed. A special thanks also to Cátia for her tireless support and encouragement.

Also a special thanks to Prof. Nuno Roma, Prof. Pedro Tomás and Prof. Nuno Neves who were always available and helpful to give the expertise and transmit the knowledge necessary to complete this work.

Last but not least, I would also like to thank all my friends and colleagues with who I had the pleasure to share this adventure with.

Thank you all.

This work was partially supported by national funds through Fundacão para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017.

Abstract

Unlimited Vector Extension (UVE) is a novel ISA extension that incorporates data streaming and scalable vectorization. To do so, it implements a data streaming engine that allows detaching the main memory access from computation, releasing some pressure from the main pipeline of the processor. This is done by statically encoding the memory access pattern of streamable loops through a set of special description instructions, to offload the corresponding address sequence generation to the streaming engine. This not only allows the computational code to be simplified by removing memory addressing instructions,but also simplifies the vectorization of the loop, in turn increasing the throughput of the processing cores. However a new extension that implements target specific instructions, it still lacks compiler support to produce target code. Accordingly, in this thesis aims at taking the first steps in the development of a compiler for UVE by instantiating a new subtarget from RISC-V's LLVM backend and creating and encoding the extension's instructions. To provide an initial integration with LLVM IR, a set of intrinsics is also proposed that match the instructions. Due to the intrinsic incompatibility between the streaming paradigm from UVE and LLVM IR SSA form, a new approach to overcome this issue is introduced based on pseudo-instructions . The new backend and LLVM IR intrinsics were evaluated with a set of benchmarks that highlight the main introduced features.

Keywords

Scalable Vector Processing; Stream Computing; LLVM IR; Compiler Backend

Resumo

Unlimited Vector Extension é uma extensão nova que incorpora fluxos de dados e vetorização escalável. Para tal, implementa uma unidade de fluxo de dados que permite desconectar acessos à memória principal da computação, aliviando alguma pressão da linha principal do processador. Isto é feito através da codificação estática de padrões de acesso à memória em ciclos que permitem fluxos de dados através de um grupo especial de instruções descriptivas, para transferir a correspondente geração de sequência de endereços para a unidade de fluxo de dados. Isto não só permite o código computacional de ser simplificado através da remoção de intruções de endereçamento à memória, mas também simplifica a vetorização do ciclo, aumentando a taxa de transferência dos núcleos de processamento. No entanto por ser uma nova extensão que implementa instruções específicas, ainda não possui suporte de compiladores para produzir o código específico. Nesse sentido, esta tese visa dar os primeiros passos no desenvolvimento de um compilador para UVE instanciando um novo sub-alvo do backend LLVM do RISC-V e criando e codificando as instruções da extensão. Para fornecer uma integração inicial com o LLVM IR, um conjunto de intrínsecas é também proposto que combina com as intruções. Devido à incompatibilidade intrínseca entre o paradigma de fluxo de dados do UVE e a forma SSA do LLVM IR, é introduzida uma nova abordagem baseada em pseudo-intruções para superar este problema. O novo backend e as intrínsecas de LLVM IR foram avaliadas com um conjunto de padrões de desempenho que destacam as principais características introduzidas.

Palavras Chave

Processamento Vectorial Escalável; Computação em Fluxo de Dados; Representação Intermédia do LLVM; Backend do Compilador

Contents

List of Figures

List of Tables

Listings

Acronyms

AST	Abstract Syntax Tree
CFG	Context Free Grammar
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSR	Control Status Registers
DAG	Directed Acyclic Graph
DCE	Dead Code Elimination
DeSC	Decoupled Supply-Compute
DFA	Determinitic Finite Automata
DLP	Data Level Parallelism
ELF	Executable and Linkable Format
HPC	High Performance Computing
IDE	Integrated Development Environment
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just-In-Time
ML	Machine Learning
MVT	Machine Value Types
NFA	Nondeterminitic Finite Automata
PBQP	Partitioned Boolean Quadratic Programming
RISC	Reduced Instruction Set Computer
SAXPY	Single-Precision A· X Plus Y

SIMD	Single Instruction Multiple Data
SLL	Shift Logical Left
SPMV	Sparse Matrix-Vector Multiply
SSA	Static Single-Assignment
SSE	Streaming SIMD Extensions
SVE	Scalable Vector Extension
TAC	Three-Address Code
UVE	Unlimited Vector Extension
VL	Vector-Lenght

1 Introduction

Contents

1.1 Motivation

Throughout the years multiple innovations have come up in the computing area to improve the performance of the processor, by using different architectures and paradigms. Some of these apply in a general way, by changing the structure completely while others are targeted for some specific purpose. One of these specific paradigms is Single Instruction Multiple Data (SIMD) [4–6], that tries to take advantage from vector-like patterns displayed by the memory accesses of some applications, such as digital signal processing and graphics processing.

SIMD units allow to fix an element size for the vector, depending on the extension, and process multiple elements while only issuing a single instruction to do it. This has clear advantages in the sense that it saves time by increasing the throughput of the processing unit.

While the use of SIMD increases the performance for some applications, it has issues. Depending on the extension, the Vector-Lenght (VL) is limited to a fixed number of bits, constraining the overall improvement that can be seen in the performance of the processor. To solve this problem the obvious solution is to eliminate the fixed limit on the VL, and that is what ARM did when they introduced Arm Scalable Vector Extension (SVE) [7], a SIMD extension to the Arm AArch64 architecture. This extension allows flexible VL implementations, that can vary from a minimum of 128 bits up to a maximum of 2048 bits, with 128-bit increments. As such, the design doesn't force the processor to implement the maximum VL and it's still able to run the same application in different implementations of SVE, without recompiling the code.

As it is a vector extension, SVE presents the largest application improvements when very large amounts of data need to processed. However it does not present a solution for memory access, where it can cause a bottleneck on the application if the data is not available for processing as it is necessary. Prefetchers partially solve this problem by speculating on what data it might be necessary on the future and fetching it ahead of time, but it is not as effective on irregular memory accesses and short memory accesses. It also never achieves a perfect prediction accuracy of the data that needs to be prefetched.

This is where the Unlimited Vector Extension (UVE) [2] presents an improvement. By combining SIMD instructions with data streaming it tries to solve both problems at once. To achieve a perfect data prefetching accuracy it describes the memory access patterns in software, allowing for that data to be streamed into a dedicated engine and used by the main processing pipeline as it is necessary.

To ease the development of new software for a new architecture it is usual to develop compiler support for that target, so that it is not necessary to write low level code. This is usually provided by the companies that also develop the architecture.

The aim of this work is to provide a LLVM backend to RISC-V's architecture extension that supports the UVE extension to be used during software development, by taking advantage of the LLVM modular structure. LLVM allows for the addition of specific UVE features while making minimal impact on the overall structure of the compiler. The new compiler extension should be able to represent UVE instructions or other form of representation, such as intrinsics, in LLVM Intermediate Representation (IR), that will latter go through the compiler pipeline and transformed into UVE target code. If possible it should also reuse some parts of other implementations that share similarities, such as Arm SVE, while also creating new solutions dedicated to the new extension.

1.2 Objectives

Given that the UVE Instruction Set Architecture (ISA) is already fully defined, a supporting compiler extension can be developed in LLVM, targeting the RISC-V architecture. This extension should give support on the backend for the new instructions as well as some way to represent such instructions in LLVM IR.

As such, the main objectives of this work are to create a fully functional UVE backend on LLVM that allows for the compilation of IR into the target assembly code, a representation that enables the use of UVE instructions inside LLVM IR and an evaluation of the implemented solution using representative benchmarks, that make use of most developed features.

1.3 Contributions

Contributions that will come out of this work are the following:

- A functional backend that can compile from IR into UVE's assembly code, using the custom instructions.
- A representation to embed UVE instructions with the rest of the LLVM IR, so that it can represent data streams and its nuances.
- One method to overcome the Single Static Assignment enforced by the LLVM IR representation and another to avoid streaming registers from being written over during register allocation.
- The successful compilation using the implemented compiler extension of a set of three benchmarks that represent the custom features.

1.4 Outline

This document is divided into four chapters. On chapter 2, a backgroud is done to explain some necessary concepts that will be used in chapters ahead related to the structure and functioning of compilers and the implementation of various vectorial extensions, culminating with UVE. Following chapter 2 is the development of compiler support, on chapter 3. During this chapter it is described the processes and methods used by LLVM to support new architectures and a solution is presented that allows to compile from an IR into assembly code. On chapter 4 an evaluation of the proposed solution is done, by compiling a set of three benchmarks that highlight the main features of the implementation. On the 5th and final chapter a conclusion is presented about the work developed and some future directions for the continuation and improvements of this work.

2 Background

Contents

2.1 Compilers

The compiler is a computer program that transforms source code, usually written in a programming language, into a certain target machine language.

To start with, it is presented a brief description of the history of the compiler and how it came to be. After that, a generic structure of a compiler is described, and the multiple stages it encompass. Lastly, the description of the LLVM Infrastructure and all the stages and methods it uses to implement a compiler, that is used within the scope of this work.

2.1.1 History of the Compiler

During the 1950's and before, programming was done on a much lower level, such as assembly language or even machine code. The main reasons for that were the constraints on memory available at the time, slow clock rates that made every single instruction precious and the amount of work needed to develop a compiler. There was no need to compile a program that was small in size, as it could be written directly in assembly, and the lack of memory available sometimes prohibited the hosting of a compiler on the machine. Also at that time programs written by hand were more optimized then the compiled versions.

In 1951 Grace Hopper first introduced the term "compiler" and in 1957 John W. Backus and his FORTRAN team developed the first commercially available compiler [8, 9].

The increase in memory availability, program complexity and processing power, as well as the cost for developing big projects during the following years led to a more widely use of compilers, to be able to program in higher-level languages. During the 1970's only critical parts of a program were written by hand, usually in assembly language as it would still lead to more optimized code [10]. For the following years, the main focus was on the performance of the resulting program and the speed it takes to compile it.

2.1.2 Structure of a Compiler

For most compilers, the structure follows the principles presented in Figure 2.1 [11]. This structure can be divided in two categories: the frontend and the backend. The frontend consists on the analysis of the source file and following decomposition into an intermediary representation that is supported by the compiler. The lexical, syntax and semantic analyzers are part of it. The remaining parts belong to the backend, where multiple processes are executed such as machine dependent and independent optimizations, resource allocation, instruction scheduling and target code generation.

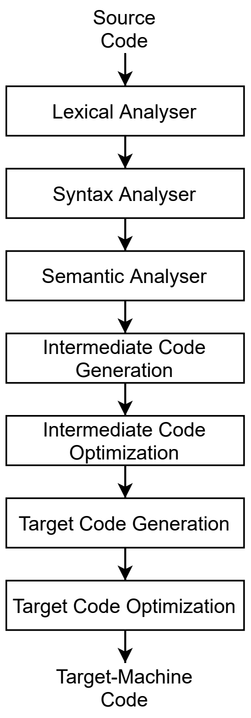


Figure 2.1: Compiler Structure.

2.1.2.A Lexical Analyser

This stage is responsible for breaking down the characters received as input from the source file into meaningful sequences called *lexemes*. This *lexemes* are then used to create *tokens* that are associated with them. Each *token* represents a logical piece of the source material, such as variable names and keywords and can contain attributes to transmit necessary extra information for the other stages of the compiler.

Lexical analysers can be implemented as finite automatas, with two main distinct types: Determinitic Finite Automata (DFA) or Nondeterminitic Finite Automata (NFA) [12]. DFAs have each possible transition to a state determined. If a DFA starts in the initial position and is given always the same input, it will always end in the same state. NFAs allow that one input might result in different outcomes, so if it starts in the same initial position and is always given as the same input it can end in different states.

As an example, in Figure 2.2 there can be identified 5 lexemes and consequently 5 tokens, 2 of them identified by the string ”i” and ”1”.

2.1.2.B Syntax Analyser

The *syntax analyser*, also known as *parser*, takes as input the tokens resulting from the previous stage and builds a tree-like object in order to give grammatical structure to the token streams. The goal is to

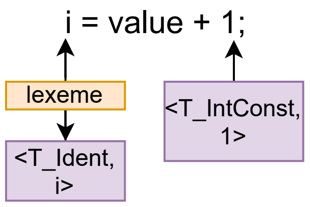


Figure 2.2: Example of Lexical Analysis.

recover the structure that was previously represented by the tokens. This trees are often represented as having the parent nodes being the operations and leaf nodes the operators and represent the productions that are used, not the order by which they occur.

Programming languages are often specified by Context Free Grammar (CFG). A CFG is a formalism for defining languages and is defined by four components: a set of non-terminals (strings that help define the language generated by the grammar), a set of terminal symbols and a set of productions (the way terminal and non-terminal symbols can be combined). These are used because various *parsers* are able to process this grammars in an efficient way. Some of the well known parsers are the LL, LALR and the Earley parser.

The LL parser uses a top-down approach to parsing: it begins with the start symbol and from there it tries to guess the productions in order to get to the input program. LALR uses the opposite approach, bottom-up. It starts with the input program and tries to reach to the start symbol. The Early parser is more complex, and can be described as a chart parser. It uses the dynamic programming approach and can parse all context-free languages, unlike the previous two.

Using the example of the tokens from Figure 2.2 it is possible to derive the tree presented in fig. 2.3.

2.1.2.C Semantic Analyser

Semantic analyses ensures that a program has a well defined meaning. It makes use of the syntax tree to make sure that the source program is consistent with the defined language semantics.

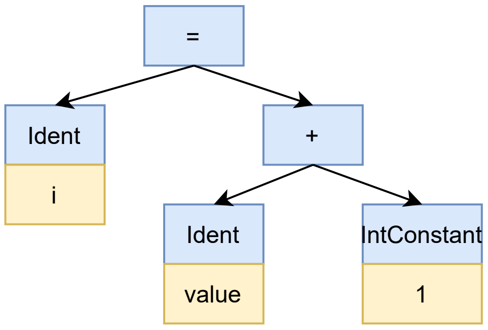


Figure 2.3: Example of Syntax Analysis.

In this step various properties of the program are verified such as the use of variables before declaration, classes not inheriting from existing classes, scope checking and type checking. After this step it is guaranteed that the user's input program is legal. Symbol tables are often used in this step to aid in scope checking.

2.1.2.D Intermediate Code Generation

After the generation of a syntactic and semantically correct tree the next step is to transform it into one or more intermediate representations that are between high-level languages and assembly or some kind of data structure. This representation usually tends to be independent from the frontend as from the backend, so if a compiler is needed for frontend **X** and backends **Y** and **Z** it is only necessary to implement the single frontend and both the backends instead of implementing a compiler for **XY** and another for **XZ**. This strategy is scales very well with an increase in the number of implementations.

Intermediate representations also allow for optimizations that are machine independent and is suitable for instruction selection and register allocation.

Some common intermediate representations include:

- **Directed Acyclic Graph (DAG)** : As the name suggests, it is a directed graph with no directed cycles. DAG's can be used as a secondary intermediate representation as it is very helpfull in the specific tasks of resources allocation and instruction selection. It is also an efficient method to identify common sub-expressions.

- **Three-Address Code (TAC)** : Code that uses this representation is characterized by it's expressions having only one or two operators, with one operation in between when there are two. This representation is most suitable for optimization and code generation.

- **Static Single-Assignment (SSA)** : This representation is similar to TAC but with two distinct features. First, it only allows for a variable with the same name to have a value assigned a single time. Second, because a variable may only have a value assigned once, to solve control flow issues where a variable would have to assume different values depending where the control comes from a ϕ-function is used to represent that.

2.1.2.E Intermediate Code Optimization

This is one of the most important stages of a compiler and a subject for a lot of studies. The previous stage is focused on translating into an intermediate stage and not on the most optimal way to do it. As a result, a lot of redundancies and overhead are introduced.

The main goals of optimization are to reduce the memory usage of a program, shorten the runtime and use as less power as possible. Optimizations can be categorized into two categories: *local optimizations* and *global optimizations*. Local optimizations aim to make a piece of code as efficient as possible inside a single *basic block*. A basic block is a piece of code that runs from the beginning to the end without any branches, except for the entrance and the exit to the block. Some local optimizations include:

- **Common Subexpression Elimination** : Eliminates expressions that recalculate unchanged values already obtained before.
- **Copy Propagation** : Replaces the uses of variable a, assigned with variable b, by variable b.
- **Dead Code Elimination** : Remove assignments that are no longer used throughout the block and are considered dead. A dead variable can originate from the two previous methods.
- **Strength Reduction** : Replacing operations by simpler ones. One example might be the replacement of a multiplication by 2 for a left shift.
- **Constant Folding** : Evaluation of expressions at compile-time, removing unnecessary computations.

Global Optimizations act on whole functions instead of a single basic block. This type of optimization is more powerful as it has a bigger picture of the whole code. Some global optimizations include:

- **Global Dead Code Elimination** : Similar to the local one, but can track variables through multiple blocks.
- **Global Constant Propagation** : Replaces each variable that is known to be a constant by the constant itself.
- **Partial Redundancy Elimination** : Eliminates expressions that are redundant on some paths but not all parts of a program.

2.1.2.F Target Code Generation

Generation of the target machine code. Allocation of resources such as registers and the cache. Because different processors have different characteristics, a good compiler needs to be able to take advantage of peculiar features in order to generate the best performing machine code. Machine specific optimizations are often done during this stage, although some are also done after at the next step. Instruction parallelism, the parallel or simultaneous execution of instructions, is also an opportunity of optimization for compilers when generating the machine code.

2.1.2.G Target Code Optimization

Finally, with the machine code ready, the last optimizations are done.

Currently most processors use some kind of pipeline as part of the architecture. As a result, the order in which the instructions are executed can cause hazards and impact the performance. This problem presents an opportunity of optimization by rescheduling instructions to minimize the time the processor is idle waiting for some operation to complete. To take advantage of multicore processors, compilers can also exploit loop parallelization.

Many caches also share a feature that tries to take advantage of code that exhibit temporal and spacial locality. Temporal locality refers to the fact that memory read recently tends to be read again. Spacial locality refers to the fact that recently read memory will likely have adjacent memory also read. An example for both cases are loops. Compilers can take advantage of this by reordering loops and doing structure peeling, that consists on dividing structures into several ones to increase the number of hits on the cache.

2.1.3 LLVM Infrastructure

The LLVM Infrastructure provides multiple tools and libraries for the implementation of a new compiler or simply create a new extension for an existing one. One feature that sets the LLVM Infrastructure apart from the other compilers is it's structure [13]. Instead of being a monolithic implementation, LLVM separates every stage into different modules and libraries. This way, if it is necessary to create a new frontend, a user only needs to be worried about translating the source file into LLVM IR, assuming that the target they want to compile to is already implemented on the backend.

2.1.3.A Frontend

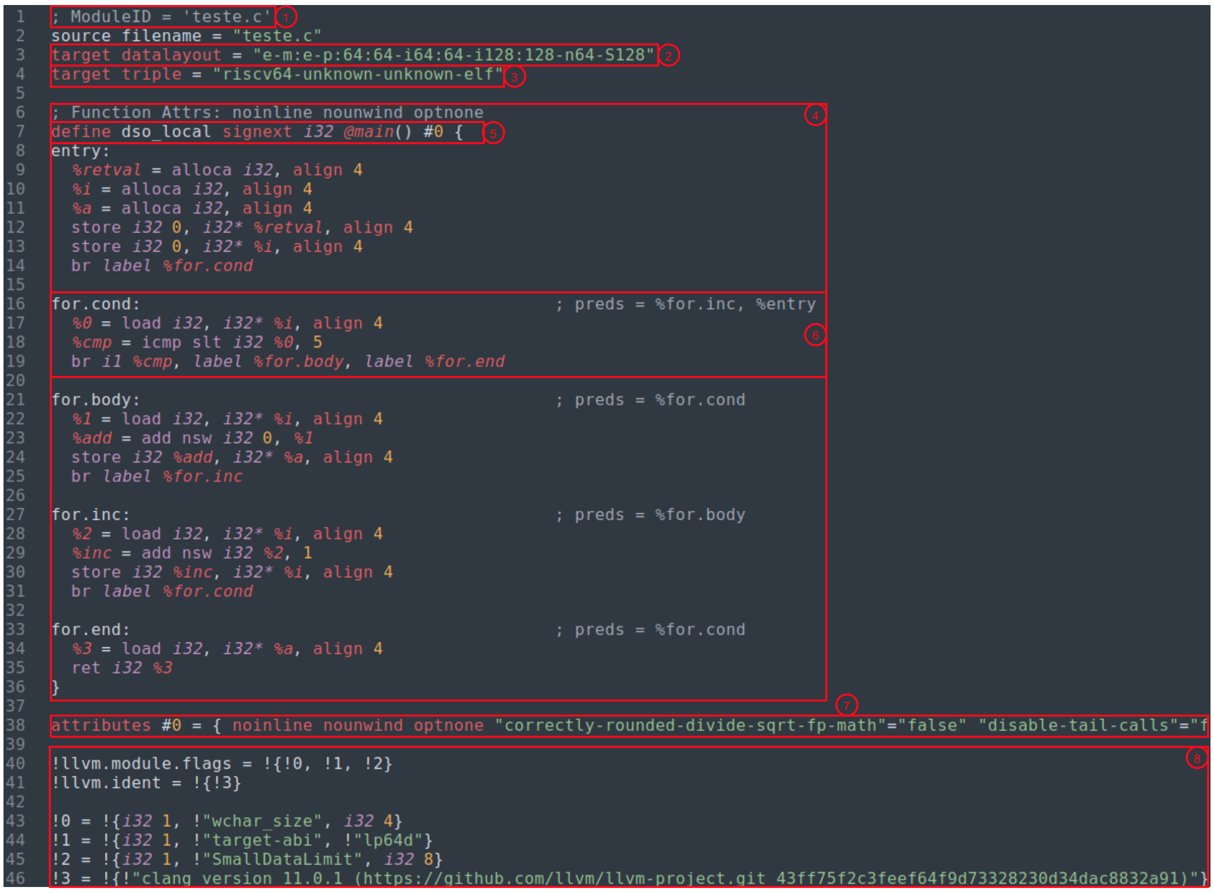
As a frontend for the C language family, the LLVM Infrastructure offers Clang. Clang was designed as a replacement for the GCC frontend. The goal was to make a frontend that was easier to integrate into an Integrated Development Environment (IDE), with a better performance and that preserved more information about the source code.

The lexical analysis done by Clang splits the source characters into tokens and creates an identifier hash table, to hold the information on the tokens. For the parser, it uses a recursive descent parser. Recursive descent parsers are a kind of top-down parsers that, as the name suggests, starts parsing from the top and reads the input from left to right. Each entity uses procedures to recursively parse the input. It is only during the syntax analysis that an Abstract Syntax Tree (AST) is built and type checking is done. At the end, the AST is converted into another intermediate representation, LLVM IR.

2.1.3.B LLVM IR

LLVM IR is an intermediate representation that separates the frontend from the backend on the Clang/LLVM compiler. This representations is it's own language and presents resemblances to the assembly language. By using a representation that is both machine and programming language independent, it enables optimizations and transformations without having to change them every time the frontend or backend changes. LLVM IR is a strongly typed SSA based representation that uses an infinite set of temporary registers and is designed to be used in three ways: as a representation in memory for the compiler in the form of C++ classes, as a binary bitcode representation that is saved on disk and can be used by Just-In-Time (JIT) compilers or in a human readable form, that resembles assembly.

LLVM IR files in the human readable form use the ".ll" extension and in each individual file it can be identified the following components, present in Figure 2.4:



```
; ModuleID = 'teste.c'
source_filename = "teste.c"
target datalayout = "e-m:e-p:64:64-i64:64-i128:128-n64-S128"
target triple = "riscv64-unknown-unknown-elf"

; Function Attrs: noinline nounwind optnone
define dso_local signext i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %i = alloca i32, align 4
  %a = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32, i32* %i, align 4
  %cmp = icmp slt i32 %0, 5
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i32, i32* %i, align 4
  %add = add nsw i32 0, %1
  store i32 %add, i32* %a, align 4
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %2 = load i32, i32* %i, align 4
  %inc = add nsw i32 %2, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %3 = load i32, i32* %a, align 4
  ret i32 %3
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="f

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 1, !"target-abi", !"lp64d"}
!2 = !{i32 1, !"SmallDataLimit", i32 8}
!3 = !{!"clang version 11.0.1 (https://github.com/llvm/llvm-project.git 43ff75f2c3feef64f9d73328230d34dac8832a91)"}
```

Figure 2.4: Example of LLVM IR.

- **Module** : *Modules* are the top level structure that hold all other LLVM IR objects of the input file. It contains functions, global variables, additional compilation information and metadata, among others. In Figure 2.4, identified by the number 1, it is the module ID.

- **Target Data Layout** : A module can specify how data is supposed to be laid out in memory, identified by the number 2 in Figure 2.4. In the example, each specification is separated by the character ”-” and the information about the specification is given after the character”:”. So the string in the example represents: little-endian form (given by ”e”); LLVM names are mangled in the output with the option Executable and Linkable Format (ELF) mangling (given by ”m:e”); the size of the pointers, in bits (given by ”p:64:64”); the size of the types, in bits (given by ”i64:64” and ”i128:128”); set of native integer widths for the target Central Processing Unit (CPU), in bits (given by ”n64”); the natural alignment of the stack, in bits (given by ”S128”).

- **Target Triple** : A string that describes the target host, identified by the number 3 in Figure 2.4. The structure of this string follows the template ”ARCHITECTURE-VENDOR-OPERATING_SYSTEM-ENVIRONMENT”, each field separated by the delimiter ”-”.

- **Functions** : A *function* is a collection of basic blocks that executes some task, like a C language function. Each basic block can have a *label* as it’s entry point (stored in a symbol table), has some instructions and finishes with a terminator instruction (branch and return), that shifts the control of the program into another basic block or just ends. Some exterior functions need to be declared with the ”declare” keyword and all functions need to be defined with the ”define” keyword. In Figure 2.4, the function is identified by the number 4, the definition of the function by number 5 and a basic block inside the function identified by number 6.

- **Variables** : A module can contain *global* and *local variables*, similarly to the C language. A global variable is available on the whole module and is defined by the ”@” character while a local variable is only available inside the function and is defined by the ”%” character. In Figure 2.4, identified by the number 5, is the global variable *main*, that is also a function.

- **Attribute Groups** : As the name suggests, they are a group of attributes that can be associated with various components, such as functions. They are identified by the number 7 in Figure 2.4.

- **Metadata** : *Metadata* allows to convey extra information to the compiler optimizer, code generator and debugging information. Identified by the number 8 in Figure 2.4.

After the generation of the LLVM IR, it begins the process of generating the code for the target machine.

2.1.3.C Backend

The backend is implemented by many LLVM libraries and takes as input the LLVM IR from the previous stage after some machine independent optimizations. The goal is to generate code for a specific target machine using a target-independent code generator. To do so it uses another intermediate representation in the form of a selection DAG by going through the following processes:

- **Generating an initial DAG**

  First the input code is transformed into a selection DAG. Instructions are mapped into DAG nodes, represented by the class *SDNode* that encapsulates the operands represented by the *SDValue* class. After the whole graph is constructed, all the nodes are linked according to the flow of the program and data dependencies. An example of the selection DAG built for the basic block identified by the number 6 in Figure 2.4 can be seen on Figure 2.5.

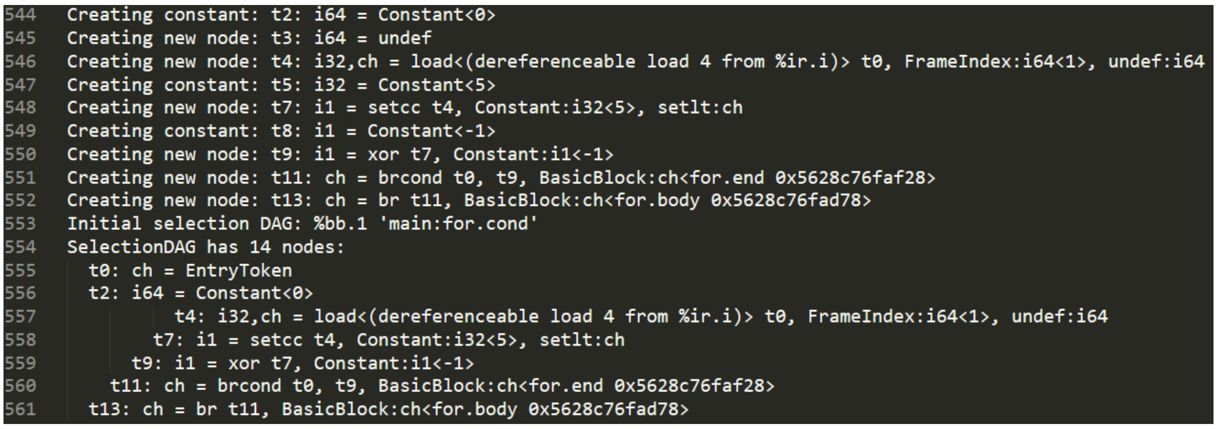
- **Optimization of the initial DAG**

  After the DAG is built, some simple optimizations are performed to simplify it. This includes recognising some instructions and combining them into others. After this optimization the example DAG from Figure 2.5 looks like Figure 2.6.

- **Legalization of selection DAG**

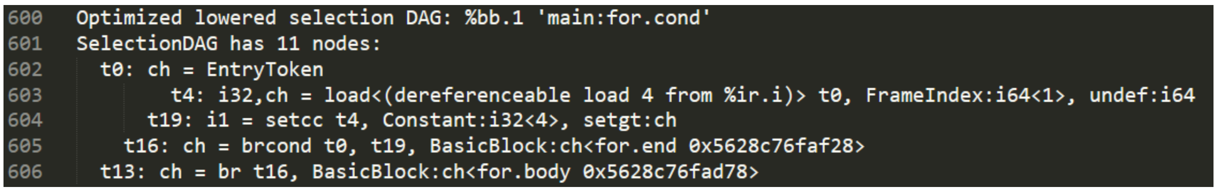
  Until now, all the steps taken are applicable for any target machine. This is the first stage where the DAG nodes are checked to see if the type and instructions used are supported by the target machine. If a type or instruction is not supported by the target machine, that node is considered illegal.

  To legalize a node, the illegal type must be lowered or expanded into other that is supported, or use a custom lowering action, allowing more options on what to do with the illegal type. Operations must be converted into others or a collection of other operations that do the same function and are supported. After the types of the operands and the instructions are legal, the node is considered legal and good to advance to the next step.



```
Creating constant: t2: i64 = Constant<0>
Creating new node: t3: i64 = undef
Creating new node: t4: i32,ch = load<(dereferenceable load 4 from %ir.i)> t0, FrameIndex:i64<1>, undef:i64
Creating constant: t5: i32 = Constant<5>
Creating new node: t7: i1 = setcc t4, Constant:i32<5>, setlt:ch
Creating constant: t8: i1 = Constant<-1>
Creating new node: t9: i1 = xor t7, Constant:i1<-1>
Creating new node: t11: ch = brcond t0, t9, BasicBlock:ch<for.end 0x5628c76faf28>
Creating new node: t13: ch = br t11, BasicBlock:ch<for.body 0x5628c76fad78>
Initial selection DAG: %bb.1 'main:for.cond'
SelectionDAG has 14 nodes:
  t0: ch = EntryToken
  t2: i64 = Constant<0>
          t4: i32,ch = load<(dereferenceable load 4 from %ir.i)> t0, FrameIndex:i64<1>, undef:i64
        t7: i1 = setcc t4, Constant:i32<5>, setlt:ch
      t9: i1 = xor t7, Constant:i1<-1>
    t11: ch = brcond t0, t9, BasicBlock:ch<for.end 0x5628c76faf28>
  t13: ch = br t11, BasicBlock:ch<for.body 0x5628c76fad78>
```

Figure 2.5: Example of selection DAG after being built.



```
Optimized lowered selection DAG: %bb.1 'main:for.cond'
SelectionDAG has 11 nodes:
 t0: ch = EntryToken
    t4: i32,ch = load<(dereferenceable load 4 from %ir.i)> t0, FrameIndex:i64<1>, undef:i64
   t19: i1 = setcc t4, Constant:i32<4>, setgt:ch
  t16: ch = brcond t0, t19, BasicBlock:ch<for.end 0x5628c76faf28>
 t13: ch = br t16, BasicBlock:ch<for.body 0x5628c76fad78>
```

Figure 2.6: Example of initial DAG after being optimized.

The legalization of the DAG is done in multiple steps, with optimization steps done in between and after it is complete.

An example of the legalization step for the operands and operations, following the example of Figure 2.6, is present in Figure 2.7. The final result of the legalization step is presented in Figure 2.8.

- **Instruction Selection**

  Here the generic target independent instructions used within LLVM IR are replaced by target specific instructions. The input is a legal DAG from the previous stage.

  On the backend, each target defines what instructions it supports and what patterns should be matched to each instruction. At this stage, the compiler takes this patterns and matches every single generic instruction or intrinsic function into an instruction supported by the target, by creating a new DAG with the native instructions. A pattern doesn't need to be just a simple one to one replacement. It can take a sequence of instructions and replace them all by another one that serves the same function. As an example, if the target supports fused multiplication and addition, it can take both instructions together and replace them by the target single instruction.

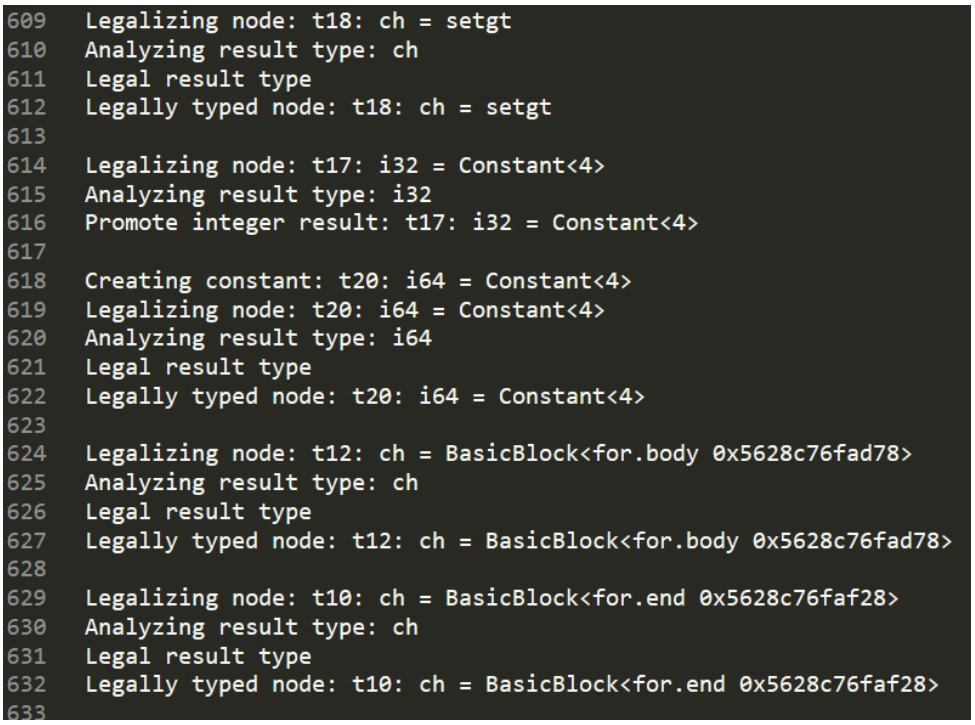
  Following the example of Figure 2.8, the instruction selection process is represented in Figure 2.9 and the resulting DAG after the selection in Figure 2.10.

- **Scheduling and emitting machine instruction**

  At this stage the compiler takes the DAG with the target machine instructions and gives them an order that depends on the target machine constraints. After all instructions are scheduled, the selection DAG is discarded and is converted into a list of machine instructions.

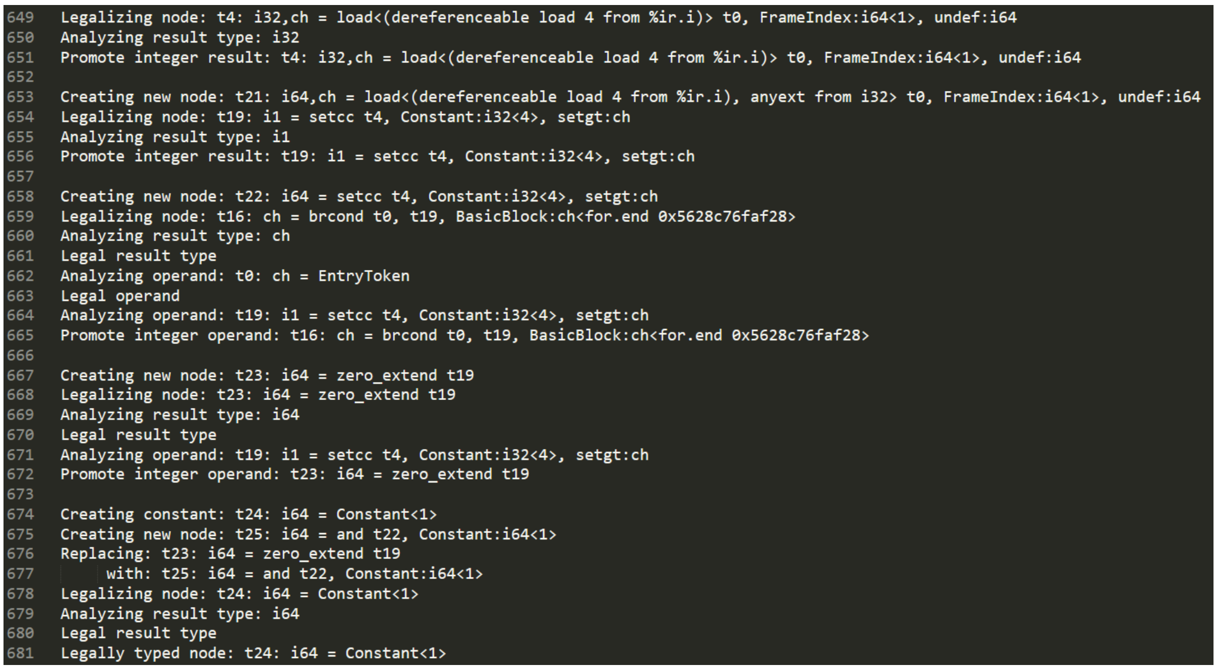
  Following the example of Figure 2.10, a schedule can be found in Figure 2.11 and the emitted machine instructions in Figure 2.12.

- **Register Allocation**



```
Legalizing node: t18: ch = setgt
Analyzing result type: ch
Legal result type
Legally typed node: t18: ch = setgt

Legalizing node: t17: i32 = Constant<4>
Analyzing result type: i32
Promote integer result: t17: i32 = Constant<4>

Creating constant: t20: i64 = Constant<4>
Legalizing node: t20: i64 = Constant<4>
Analyzing result type: i64
Legal result type
Legally typed node: t20: i64 = Constant<4>

Legalizing node: t12: ch = BasicBlock<for.body 0x5628c76fad78>
Analyzing result type: ch
Legal result type
Legally typed node: t12: ch = BasicBlock<for.body 0x5628c76fad78>

Legalizing node: t10: ch = BasicBlock<for.end 0x5628c76faf28>
Analyzing result type: ch
Legal result type
Legally typed node: t10: ch = BasicBlock<for.end 0x5628c76faf28>
```

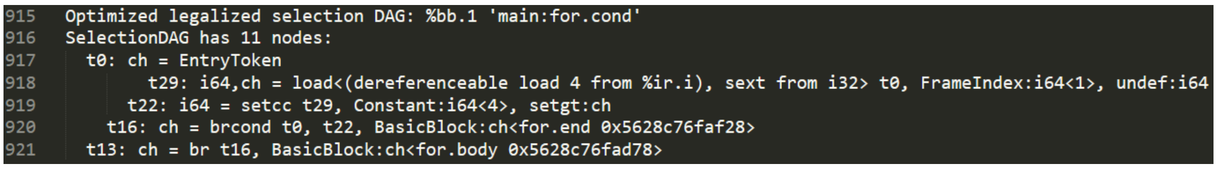
(a) Operators Legalization.



```
Legalizing node: t4: i32,ch = load<(dereferenceable load 4 from %ir.i)> t0, FrameIndex:i64<1>, undef:i64
Analyzing result type: i32
Promote integer result: t4: i32,ch = load<(dereferenceable load 4 from %ir.i)> t0, FrameIndex:i64<1>, undef:i64

Creating new node: t21: i64,ch = load<(dereferenceable load 4 from %ir.i), anyext from i32> t0, FrameIndex:i64<1>, undef:i64
Legalizing node: t19: i1 = setcc t4, Constant:i32<4>, setgt:ch
Analyzing result type: i1
Promote integer result: t19: i1 = setcc t4, Constant:i32<4>, setgt:ch

Creating new node: t22: i64 = setcc t4, Constant:i32<4>, setgt:ch
Legalizing node: t16: ch = brcond t0, t19, BasicBlock:ch<for.end 0x5628c76faf28>
Analyzing result type: ch
Legal result type
Analyzing operand: t0: ch = EntryToken
Legal operand
Analyzing operand: t19: i1 = setcc t4, Constant:i32<4>, setgt:ch
Promote integer operand: t16: ch = brcond t0, t19, BasicBlock:ch<for.end 0x5628c76faf28>

Creating new node: t23: i64 = zero_extend t19
Legalizing node: t23: i64 = zero_extend t19
Analyzing result type: i64
Legal result type
Analyzing operand: t19: i1 = setcc t4, Constant:i32<4>, setgt:ch
Promote integer operand: t23: i64 = zero_extend t19

Creating constant: t24: i64 = Constant<1>
Creating new node: t25: i64 = and t22, Constant:i64<1>
Replacing: t23: i64 = zero_extend t19
     with: t25: i64 = and t22, Constant:i64<1>
Legalizing node: t24: i64 = Constant<1>
Analyzing result type: i64
Legal result type
Legally typed node: t24: i64 = Constant<1>
```

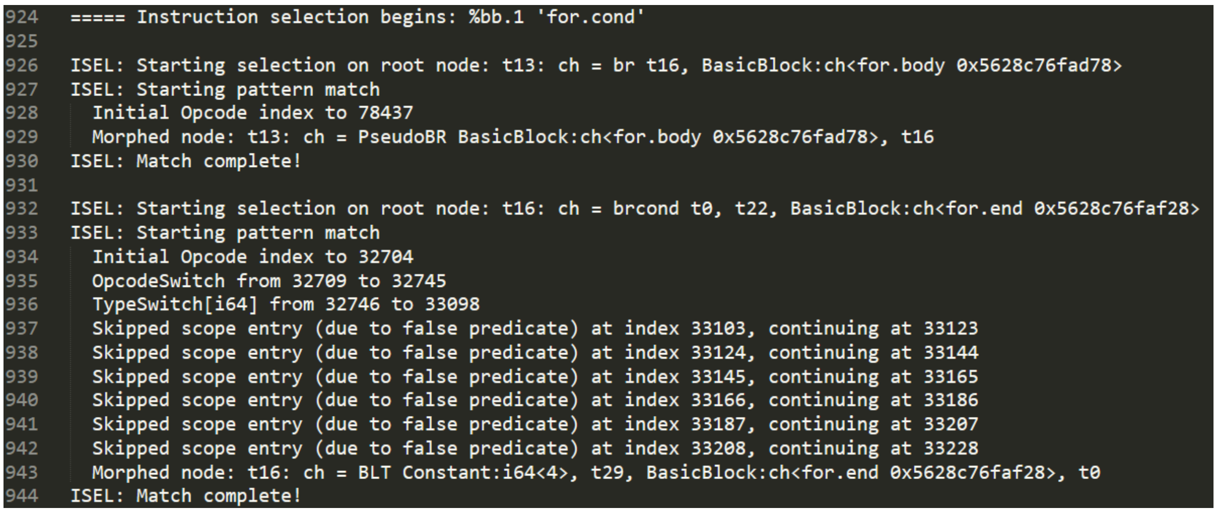
(b) Operation Legalization.

Figure 2.7: Example of DAG legalization.



```
Optimized legalized selection DAG: %bb.1 'main:for.cond'
SelectionDAG has 11 nodes:
  t0: ch = EntryToken
        t29: i64,ch = load<(dereferenceable load 4 from %ir.i), sext from i32> t0, FrameIndex:i64<1>, undef:i64
      t22: i64 = setcc t29, Constant:i64<4>, setgt:ch
    t16: ch = brcond t0, t22, BasicBlock:ch<for.end 0x5628c76faf28>
  t13: ch = br t16, BasicBlock:ch<for.body 0x5628c76fad78>
```

Figure 2.8: Example of final DAG after legalization.



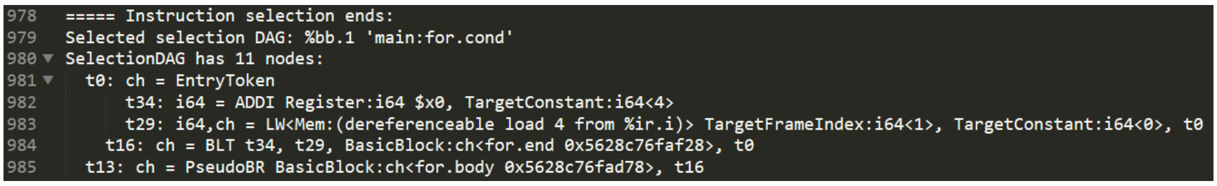
```
===== Instruction selection begins: %bb.1 'for.cond'

ISEL: Starting selection on root node: t13: ch = br t16, BasicBlock:ch<for.body 0x5628c76fad78>
ISEL: Starting pattern match
  Initial Opcode index to 78437
  Morphed node: t13: ch = PseudoBR BasicBlock:ch<for.body 0x5628c76fad78>, t16
ISEL: Match complete!

ISEL: Starting selection on root node: t16: ch = brcond t0, t22, BasicBlock:ch<for.end 0x5628c76faf28>
ISEL: Starting pattern match
  Initial Opcode index to 32704
  OpcodeSwitch from 32709 to 32745
  TypeSwitch[i64] from 32746 to 33098
  Skipped scope entry (due to false predicate) at index 33103, continuing at 33123
  Skipped scope entry (due to false predicate) at index 33124, continuing at 33144
  Skipped scope entry (due to false predicate) at index 33145, continuing at 33165
  Skipped scope entry (due to false predicate) at index 33166, continuing at 33186
  Skipped scope entry (due to false predicate) at index 33187, continuing at 33207
  Skipped scope entry (due to false predicate) at index 33208, continuing at 33228
  Morphed node: t16: ch = BLT Constant:i64<4>, t29, BasicBlock:ch<for.end 0x5628c76faf28>, t0
ISEL: Match complete!
```

Figure 2.9: Example of the instruction selection process.

Up until now it was assumed that there could be as many virtual registers as necessary. However, real machine only have a limited number of physical registers. During this stage, the compiler assigns physical registers to the virtual register. If there are not enough physical registers to assign, the compiler moves the value of one of the registers to the main memory, in a process called *spilling*.

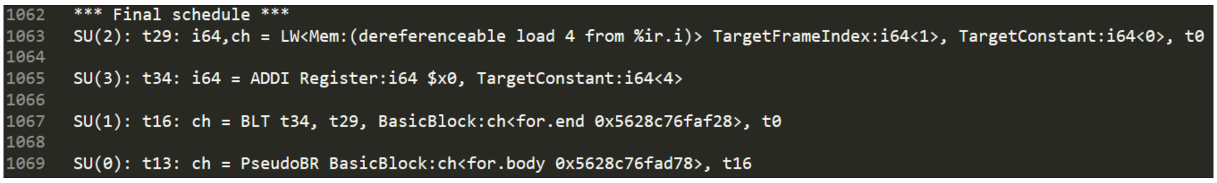
LLVM allocates registers in two different ways: *direct mapping* and *indirect mapping*. In the first case, the classes associated with the physical registers and machine operands are responsible for the allocation. On the second case, the class associated with the virtual registers are responsible for the allocation. LLVM also makes available four register allocation techniques: *Fast*, where it allocates register on a basic block level and tries to keep the values on the registers to reuse them; *Basic*, an incremental approach to register allocation, that makes use of live range analysis (where a variable has a live value during the execution of the program) and spill weight for prioritization of virtual registers; *Greedy*, an optimized version of the *Basic* method and incorporates a global live range splitting; *PBQP*, a Partitioned Boolean Quadratic Programming (PBQP) based register allocator, that represents the allocation of registers as a PBQP problem and solves it using a PBQP solver.

Following the example of Figure 2.12, the allocation of registers using the greedy method is



```
===== Instruction selection ends:
Selected selection DAG: %bb.1 'main:for.cond'
SelectionDAG has 11 nodes:
  t0: ch = EntryToken
      t34: i64 = ADDI Register:i64 $x0, TargetConstant:i64<4>
      t29: i64,ch = LW<Mem:(dereferenceable load 4 from %ir.i)> TargetFrameIndex:i64<1>, TargetConstant:i64<0>, t0
    t16: ch = BLT t34, t29, BasicBlock:ch<for.end 0x5628c76faf28>, t0
  t13: ch = PseudoBR BasicBlock:ch<for.body 0x5628c76fad78>, t16
```

Figure 2.10: Example of final DAG after instruction selection.



```
*** Final schedule ***
SU(2): t29: i64,ch = LW<Mem:(dereferenceable load 4 from %ir.i)> TargetFrameIndex:i64<1>, TargetConstant:i64<0>, t0

SU(3): t34: i64 = ADDI Register:i64 $x0, TargetConstant:i64<4>

SU(1): t16: ch = BLT t34, t29, BasicBlock:ch<for.end 0x5628c76faf28>, t0

SU(0): t13: ch = PseudoBR BasicBlock:ch<for.body 0x5628c76fad78>, t16
```

Figure 2.11: Example of instruction scheduling.

represented in Figure 2.13.

- **Code emission**

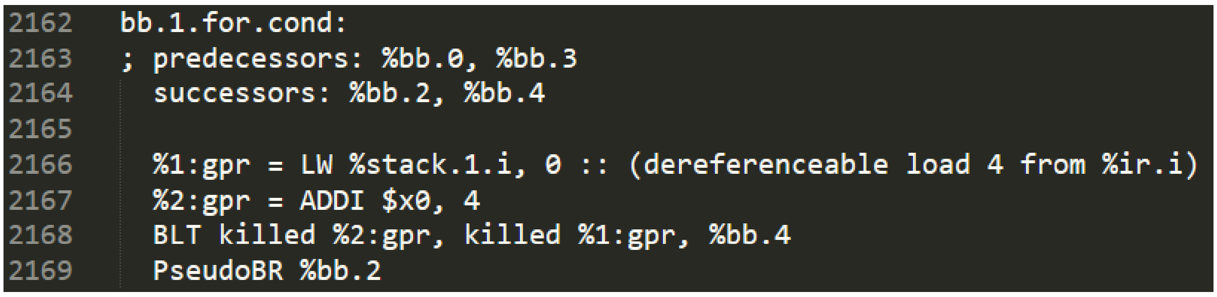
  The final step is to emit the machine code into a binary format or assembly. The compiler takes as input the *Machine Instructions* from the previous steps and lowers them into *Machine Code Instructions*, used by the *Machine Code* layer. One of the advantages LLVM has over traditional compilers is that, usually, traditional compilers output only the assembly code and then need an external assembler to produce the object file. However, LLVM uses its own assembler so it can print directly into binary form and with some wraps, to object file. This features allows to save time and guarantees that the output in textual format will be the same as the output in binary formats.

2.2 Vector Extensions

In this section it will be given an overview of the processing strategy SIMD and some popular vectorial extensions, such as the RISCV ”V” vector extension and the Arm SVE. It will also be discussed the pros and cons of using such strategies and extensions. To present an alternative, it will be introduced the streaming paradigm and UVE, an experimental ISA extension that makes use of data streams.

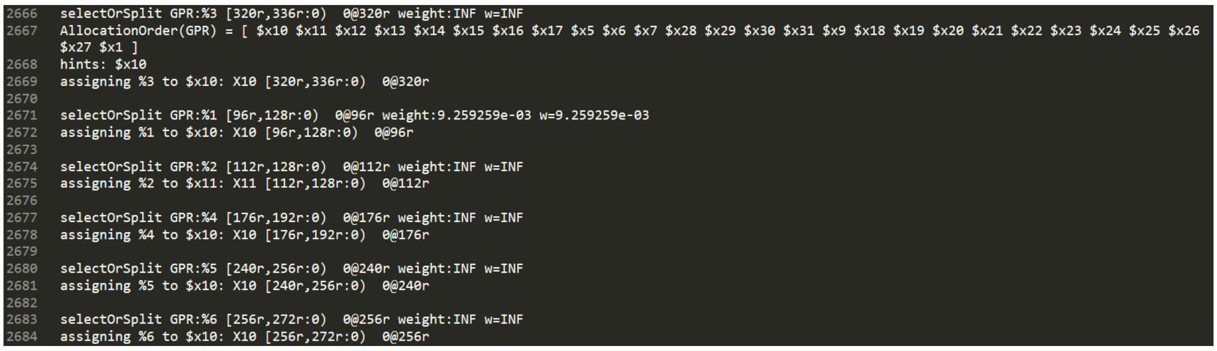
2.2.1 SIMD and Scalable Vectorial Extensions

The use of SIMD based extensions and scalable vectorial extensions aim to take advantage of Data Level Parallelism (DLP) by processing multiple data while only emitting one single processing instruction. To do so, the data to be processed if aggregated into vectors, that then go through some processing unit, executing the same operations on all the data.



```
bb.1.for.cond:
; predecessors: %bb.0, %bb.3
  successors: %bb.2, %bb.4

  %1:gpr = LW %stack.1.i, 0 :: (dereferenceable load 4 from %ir.i)
  %2:gpr = ADDI $x0, 4
  BLT killed %2:gpr, killed %1:gpr, %bb.4
  PseudoBR %bb.2
```

Figure 2.12: Example of emitting machine instruction.



```
selectOrSplit GPR:%3 [320r,336r:0)  0@320r weight:INF w=INF
AllocationOrder(GPR) = [ $x10 $x11 $x12 $x13 $x14 $x15 $x16 $x17 $x5 $x6 $x7 $x28 $x29 $x30 $x31 $x9 $x18 $x19 $x20 $x21 $x22 $x23 $x24 $x25 $x26
$x27 $x1 ]
hints: $x10
assigning %3 to $x10: X10 [320r,336r:0)  0@320r

selectOrSplit GPR:%1 [96r,128r:0)  0@96r weight:9.259259e-03 w=9.259259e-03
assigning %1 to $x10: X10 [96r,128r:0)  0@96r

selectOrSplit GPR:%2 [112r,128r:0)  0@112r weight:INF w=INF
assigning %2 to $x11: X11 [112r,128r:0)  0@112r

selectOrSplit GPR:%4 [176r,192r:0)  0@176r weight:INF w=INF
assigning %4 to $x10: X10 [176r,192r:0)  0@176r

selectOrSplit GPR:%5 [240r,256r:0)  0@240r weight:INF w=INF
assigning %5 to $x10: X10 [240r,256r:0)  0@240r

selectOrSplit GPR:%6 [256r,272r:0)  0@256r weight:INF w=INF
assigning %6 to $x10: X10 [256r,272r:0)  0@256r
```

Figure 2.13: Example of register allocation using the greedy method.

However, the way it is used to achieve such goals is different from strategy to strategy and from extension to extension. On the following chapters, it is given a description of the strategy used by SIMD and scalable vectorial extensions and some of the extensions designed using such strategies.

2.2.1.A SIMD

SIMD extensions make use of fixed length vectors to process multiple data with a single instruction, taking advantage of DLP. In order to do so, the processing units must be able to handle multiple data and are referred to as SIMD units. To take advantage of this units it is also important to combine multiple loads and stores accesses. This can be achieved with contiguous memory accesses and with arrays that are aligned in memory. This solutions are great for specific problems that can be vectorized and have a huge amount of data to be processed but for other applications that don't share these characteristics it can be detrimental. Because it uses larger registers it also consumes more energy.

Two extensions that make use of SIMD instruction are the x86 SSE/AVX extension by Intel [4, 5] and the NEON extension by Arm [6]. This two extensions are featured in two different instruction sets architectures, the first in a Complex Instruction Set Computer (CISC) based processor and the second in a Reduced Instruction Set Computer (RISC) based processor. Streaming SIMD Extensions (SSE) and AVX are both designed by Intel for the x86 instruction set architecture. The first SIMD based extension released by Intel that started it all was the MMX instruction set. This extension supported registers 64 bits wide but could not operate on floating point and SIMD data at the same time, because it reused the x87 floating point registers. To improve on this extension, in 1999, Intel released their next extension, SSE. This extension had a new 128 width bit independent register file that could already support floating point operations. The next iteration on this extension came in the form of SSE2, SSE3 and SSE4, with the addition of double-precision floating point operations and a lot of supporting instructions. In 2011, Intel released AVX, that widened the data path from 128 bit registers to 256 and three-operand SIMD instructions. Following that it came AVX2 and, the latest extension, AVX-512, that again extended the registers from 256 bits to 512 bits.

The NEON extension is designed by Arm and is intended to accelerate multimedia applications, signal and video processing, among others. It is composed of 32 64-bit separate registers capable of storing 64 bits (doubleword) and 128 bits (quadword) and NEON instructions support 8-bit, 16-bit, 32-bit and 64-bit signed and unsigned integers as also 32-bit single-precision floating point elements. These instructions perform memory accesses, data copying between NEON and general purpose registers, data type conversion and data processing.

By comparing both approaches, Intel gets the wider registers with the AVX-512 extension, capable of storing 512 bits, in comparison to Arm's NEON extension of only 128 bits, leading to being able to operate on more data at the same time, and thus reducing the number of clock cycles necessary and execution time. However that comes at a cost, by consuming more energy and occupying more space. SIMD extensions provides a benefit by operating on large amounts of data at the same time but it still has some problems. The one that stands out immediately is the size of the register being a fixed number. This forces developers to design implementations for many different sized registers architectures, otherwise the application may not have the same performance by having vectors too wide or to narrow, leading to a waste of vector space or the emission of too many load and store instructions, respectively.

2.2.1.B Scalable Vectorial Extensions

Scalable vector extensions appear as an attempt to mitigate the disadvantages of the SIMD based extensions, by removing the fixed limit on the size of the vectors. This strategy removes the necessity of designing different implementations depending on the target's vector size and also removes the necessity to worry about some edge cases, with only having to scale the vector in order to solve them. The main extensions that share this ideology are RISC-V ”V” extension [14] and Arm's SVE [7].

The RISC-V ”V” vector extension defines that the size of a vector element, ELEN, must be higher or equal to 8 bits and a power of 2, and the size of a vector register, VLEN, must be higher or equal to the size ELEN and should also be a power of 2. This extension adds 32 vector registers and 7 Control Status Registers (CSR), that allow to control the vector starting position, length, data type register and vector register length in run-time. This extension also supports vector masking, that allows finer control over which elements inside the vectors are processed.

Arm's SVE extension was designed with High Performance Computing (HPC) and Machine Learning (ML) applications in mind. This extension includes 32 new scalable vector registers with their length being implementation dependent, from 128 to 2048 bits, in increments of 128 bits. The data elements can range from 8 bits to 16, 32 and 64 bits. It also features 16 scalable predicate registers, that allow per lane predication and a more selective choice of what data elements are going to be processed, same as RISC-V "V" extension, although only 8 are used for general memory and arithmetic operations, as a way to mitigate the predicate register pressure observed in other predicate-centric architectures. It also features a special-purpose first-fault register. Following SVE, it came SVE2, that shares most of functionality with SVE but it extends the instruction set to cover more data-processing domains, such as computer vision, multimedia and web serving.

Both of these extension function in a similar way; they both need to define the vector and elements length, a predicate register, and then operate, with a single instruction, over this vectors. RISC-V's "V" extension is able to configure the scalable vectors in a more refined way comparing to Arm's SVE extension. The use of scalable vectors eases the problem that was the fixed limit of SIMD based extensions, and also presents better results when dealing with edge case scenarios. However, both these extension still need to rely on instructions to calculate memory accesses and to emit loop iteration control instructions, adding on to the pipeline. Both Arm's SVE and RISC-V "V" extension are already supported in LLVM [15–17].

2.3 Unlimited Vector Extension

UVE is an experimental ISA extension proposed by Domingos et al. [2] to overcome the disadvantages described in the previous scalable extensions. In order to do so, it implements a streaming engine to decouple the memory accesses from the main processing pipeline and makes use of data streams. To implement the data streams, UVE needs to be able to represent various memory accesses, from simple linear iterations to indirect memory accesses.

As implementing compiler support for this extension is the main focus of this work, UVE will be described with great detail on the next sections, to be able to comprehend some of the decisions done on the implementation.

2.3.1 Memory Access Description

In order for UVE to implement streams, it needs to be able to describe them. To describe a stream, UVE makes use of memory pattern description through descriptors. The base of the descriptor is given by

Neves et al. [1,18,19] works, where it is specified that by using a combination of descriptors any regular memory access pattern can be represented. The address sequence can be represented as an affine function, given by:

$$y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times stride_k, \text{ with } x_k \in [\alpha_k, \beta_k],\ X = \{x_0 \cdots x_{dim_y}\} \tag{2.1}$$

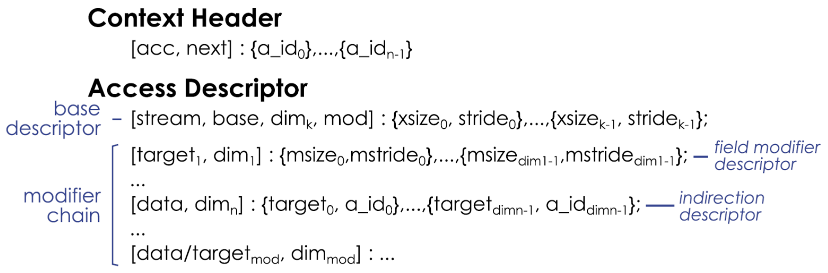
where each stream access $y(X)$ is described by the sum of the base address (y_{base}) with dim_y pairs, described by their size (x_k) and stride ($stride_k$). This representation is capable of describing any pattern but can introduce a huge amount of descriptors.

A form of describing memory accesses that circumvents this problem is proposed by Neves et al. in Figure 2.14.

This specification is composed of a context header, a base descriptor and a modifier chain. The base descriptor is composed of a stream identification ($stream$), memory base address ($base$), the number of dimensions it has (dim_y), the number of modifiers (mod) and the description of each dimension, given by its size ($xsize$) and stride ($stride$). The encoded pairs configured more to the right (with higher index) represent higher or outermost dimensions. This base description allows for the representation of any N-dimension memory description but is would use too many dimensions to do so.

The modifier chain allows to represent non-linear memory patterns, and consequently decrease the number of descriptors necessary to represent such memory accesses. Modifiers are applied to the base descriptor every time after a complete iteration of the corresponding dimension pair it is associated with. Modifiers descriptors are divided into two categories: field modifier descriptors and indirect descriptors.

Field modifier descriptors are as described in Figure 2.14 and contain the target of the modifier ($target_{mod}$), the dimension of the modifier (dim_{mod}) and then a pair that includes the size of the modifier ($msize$) and its stride ($mstride$). A use of this modifier is exemplified in Figure 2.15. In this example the source code pretends to make a triangular matrix memory access. To implement this pattern as a descriptor, a base descriptor is defined with two dimensions, the first by $\{1, 1\}$ that iterates over the elements on a row and the second by $\{N, N\}$ that jumps over the rows N times. After that, a field modifier



Context Header

[acc, next] : {a_id$_0$},...,{a_id$_{n-1}$}

Access Descriptor

[stream, base, dim$_k$, mod] : {xsize$_0$, stride$_0$},...,{xsize$_{k-1}$, stride$_{k-1}$};

[target$_1$, dim$_1$] : {msize$_0$,mstride$_0$},...,{msize$_{dim1-1}$,mstride$_{dim1-1}$};

...

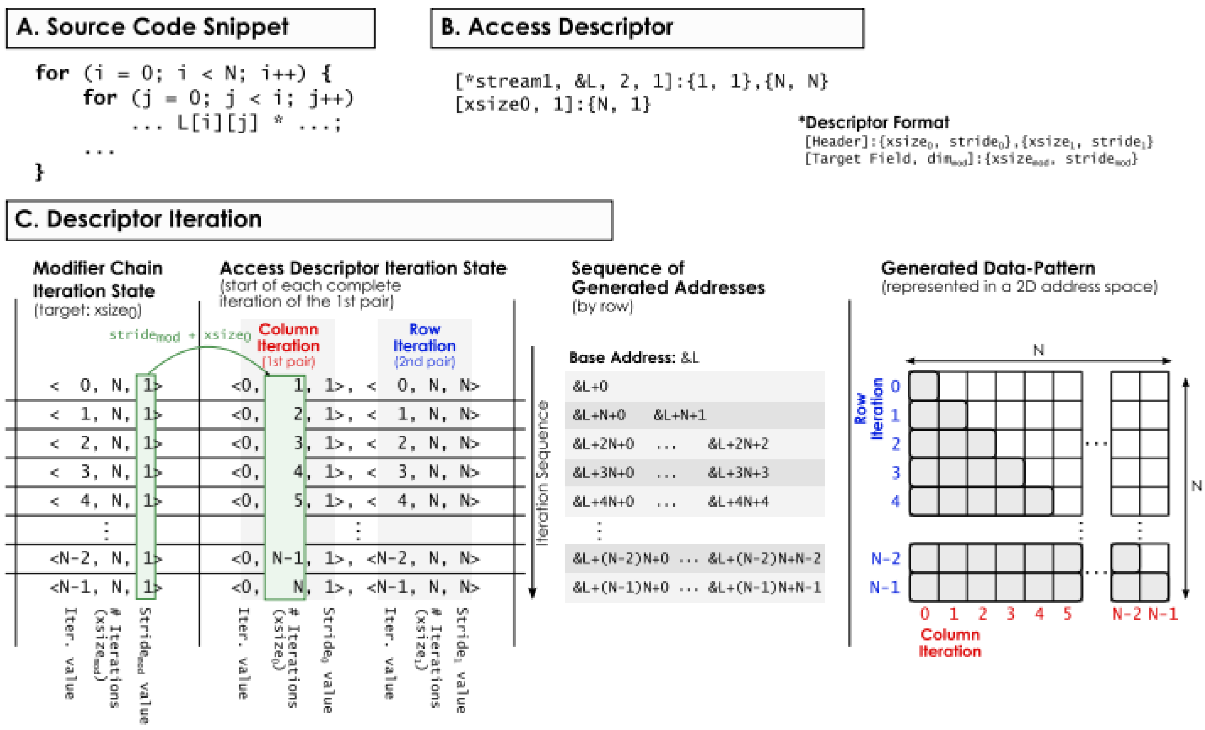
[data, dim$_n$] : {target$_0$, a_id$_0$},...,{target$_{dimn-1}$, a_id$_{dimn-1}$};

...

[data/target$_{mod}$, dim$_{mod}$] : ...

Figure 2.14: Descriptor specification proposed by Neves et al. [1].

descriptor is defined targeting $xsize0$ (the size of dimension 1 from the base descriptor) with dimension 1, characterized with size N and stride 1. This modifier will change the size of the first dimension on the base descriptor every full iteration, increasing it by 1 unit, leading to a triangular matrix access.



A. Source Code Snippet

```
for (i = 0; i < N; i++) {
    for (j = 0; j < i; j++)
        ... L[i][j] * ...;
    ...
}
```

B. Access Descriptor

```
[*stream1, &L, 2, 1]:{1, 1},{N, N}
[xsize0, 1]:{N, 1}
```

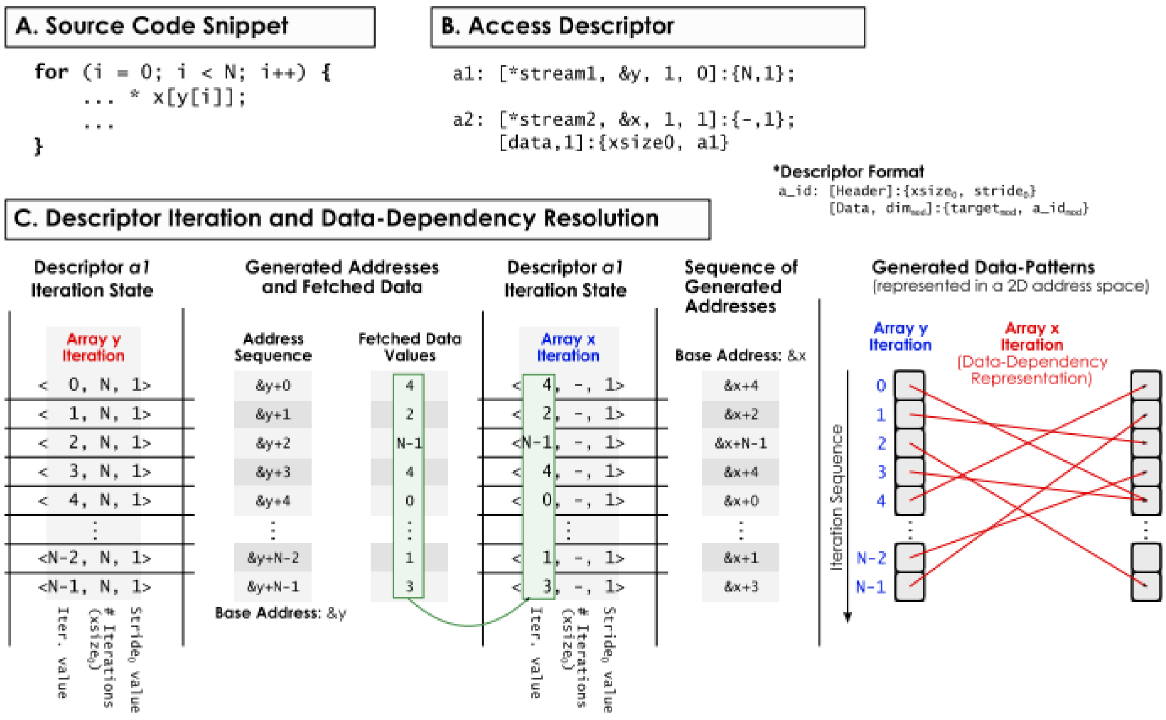
*Descriptor Format

```
[Header]:{xsize0, stride0},{xsize1, stride1}
[Target Field, dimmod]:{xsizemod, stridemod}
```

Modifier Chain	Access Descriptor	Generated Addresses
< 0, N, 1>	<0, 1, 1>, < 0, N, N>	&L+0
< 1, N, 1>	<0, 2, 1>, < 1, N, N>	&L+N+0 &L+N+1
< 2, N, 1>	<0, 3, 1>, < 2, N, N>	&L+2N+0 ... &L+2N+2
< 3, N, 1>	<0, 4, 1>, < 3, N, N>	&L+3N+0 ... &L+3N+3
< 4, N, 1>	<0, 5, 1>, < 4, N, N>	&L+4N+0 ... &L+4N+4
⋮	⋮	⋮
<N-2, N, 1>	<0, N-1, 1>, <N-2, N, N>	&L+(N-2)N+0 ... &L+(N-2)N+N-2
<N-1, N, 1>	<0, N, 1>, <N-1, N, N>	&L+(N-1)N+0 ... &L+(N-1)N+N-1

Figure 2.15: Access Descriptor encoding a triangular matrix access by Neves et al. [1].

Indirect descriptors can be used to encode data dependencies between descriptors. To describe one, it is necessary to specify the $data$ label, the dimension of the modifier (dim_{mod}) and the pairs composed by the target ($target$) and stream descriptor (a_id). The stream descriptor in the pair emphasises the data dependency between the descriptors. An example of this modifier is given in Figure 2.16. In this example, on the source code snippet, the y variable is being iterated every loop and used as an index for the x variable, creating a data dependency between y and x. So, the y variable can be represented as a descriptor by simply defining its only dimension, of size N and stride 1. To describe the x variable it is necessary to first define a descriptor that has a dummy dimension pair, of undefined size and stride 1. This is because the size is going to be defined by the size of the descriptor used in the modifier. The modifier defines a pair, targeting the $xsize0$ field and the descriptor $a1$, referring to the variable y. With this definition, every time the descriptor $a1$ produces a value, that value will be used as an offset by descriptor $a2$ to create indirect memory accesses.



```
for (i = 0; i < N; i++) {
    ... * x[y[i]];
    ...
}
```

```
a1: [*stream1, &y, 1, 0]:{N,1};

a2: [*stream2, &x, 1, 1]:{-,1};
    [data,1]:{xsize0, a1}
```

```
a_id: [Header]:{xsize0, stride0}
      [Data, dimmod]:{targetmod, a_idmod}
```

Figure 2.16: Access Descriptor encoding an indirect memory indexation by Neves et al. [1].

2.3.2 UVE Streams

In UVE, a stream is a continuous flow of data that makes use of descriptors to describe memory patterns for its configuration and manipulation. The descriptors used in UVE take inspiration from the ones described previously but are made simpler for an easier integration with for loops, the main target this extension.

The linear patterns used in UVE resemble the base descriptor in Figure 2.14, by using three parameters to describe a dimension: offset, equivalent to $base$ from the base descriptor and size and stride, that formed a pair for defining dimensions on the base descriptor. It is also necessary to detail the data-type, to know when accessing from memory how many bits can be read. The conjunction of this parameters, as before, defines a descriptor, that is a representation of a stream. To add more dimensions to the previously defined descriptor, one only needs to add a new descriptor of top, by following the same procedure as for the first one.

In order to represent more complex memory access patterns, UVE also makes use of modifiers. The modifier must be associated to a single dimension. Each one is characterized by the target the modifier wants to change (a choice between offset, size and stride from the aforementioned linear pattern), the behaviour it wants to implement (increment and decrement), the amount it will increase or decrease the targeted parameter and the size of the modifier (the number of iterations it will perform this modification).

Lastly, to represent indirect accesses UVE uses the same approach as the previous modifiers, however, since indirect modifiers represent a data dependency between two variables, instead of selecting the amount it will modify the targeted dimension this data will come from a stream. As such, the definition of an indirect modifier is done by selecting the target it will modify, the behaviour of such modification (add, subtract, increment, decrement and set), the source stream for the implementation of such behaviours and the size of the modifier, with the same purpose as in the last one.

A summary of the descriptors and modifiers detailed previously is shown in Figure 2.17.

Lastly, to be able to use descriptors with multiple dimensions and modifiers associated with them UVE implements a list. Each node of this list can have, at most, one dimension and one modifier and the list is built from the inner most loop, as the head of the list, to the outer most loop. This list is connected and directed, to imply order between the nodes. A way to build such ordered list can be done by issuing every single dimension and modifier with one instruction, building upon the latter one.

2.3.3 UVE Instruction Set Architecture

After defining the concept of descriptor, and how they are used to describe a stream, the UVE instructions make use of them to define their architecture and the instructions supported in the ISA.

UVE is implemented as an extension to RISC-V, leveraging the fact it is an open-source architecture. It makes use of a streaming engine as a way to decouple the memory accesses from the main processing pipeline. This way, it can save time by letting the streaming unit deal with some memory operations and removing some memory instructions from the main processing unit.

2.3.3.A Register File

It contains 32 vectorial registers, named from ”u0” to ”u31”, where each vectorial register doesn’t have a maximum size but it has a minimum size, value defined by the maximum element width implemented by

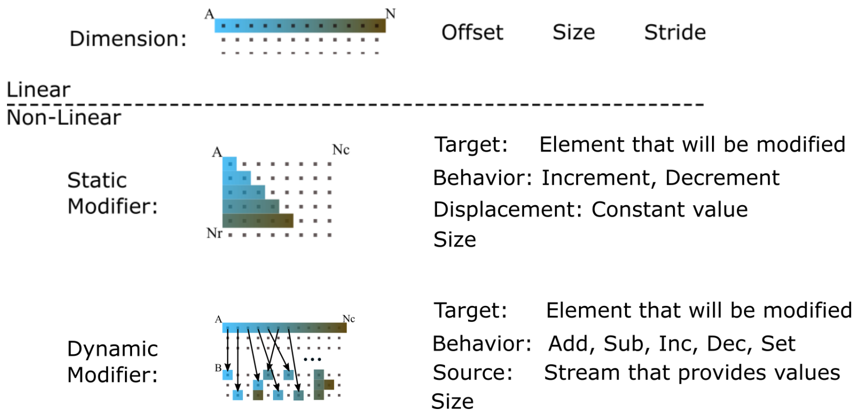

Dimension:
A
N
Offset
Size
Stride
Linear
Non-Linear
Static
Modifier:
A
Nc
Nr
Target: Element that will be modified
Behavior: Increment, Decrement
Displacement: Constant value
Size
Dynamic
Modifier:
A
Nc
B
Target: Element that will be modified
Behavior: Add, Sub, Inc, Dec, Set
Source: Stream that provides values
Size

Figure 2.17: Descriptors summary containing linear descriptors and dynamic modifiers by Domingos et al. [2].

the architecture. The available vector element widths are byte (8-bit), half-word (16-bit), word (32-bit) and double-word (64-bit). As such, the minimum value for the vectorial register size is 64 bits. This extension also makes use of CSR, that can be used to discover the vector maximum length and configure the working length. Additionally, each vector must also hold control bits to define the number of elements in the vector that are valid. A representation of a UVE vectorial register is given in Figure 2.18.

The registers used by UVE as vectorial registers have no differences from streaming registers, instead using an implicit definition of streaming registers. To configure one as a stream, an instruction must be issued with that effect and the streaming engine must keep a tab on the active streams. Reading or writing to a register configured as a stream is the same as reading or writing to the stream itself, function done also by the streaming engine.

UVE also supports a predication mechanism similar to the ones used by the other scalable vector extensions. The register file contains 16 predication registers , from ”p0” to ”p15”, but, similarly to Arm’s SVE extension, only 8 are used to perform memory and arithmetic operations. As a particularity, the ”p0” register is always set to 1, enabling all lanes to execute. When the predication value is 0, the operation is not performed on that element index. Instead, the value on that index in the output vector remain unchanged.

2.3.3.B Instructions overview

This new extension supports a total of 82 instructions with around 450 variants, so, instead of detailing each one, it will only be presented a instruction to explain its format and how it works in every group of instructions, given that the remaining instructions in that group follow a similar pattern and can be deduced easily.

Scalable vectors do not contain the data-type of the elements present in vectors, so this job is delegated to the instructions, that must explicitly declare which data-type are they going to operate on.

First, the arithmetic instructions. UVE supports arithmetic instructions between scalable vectors

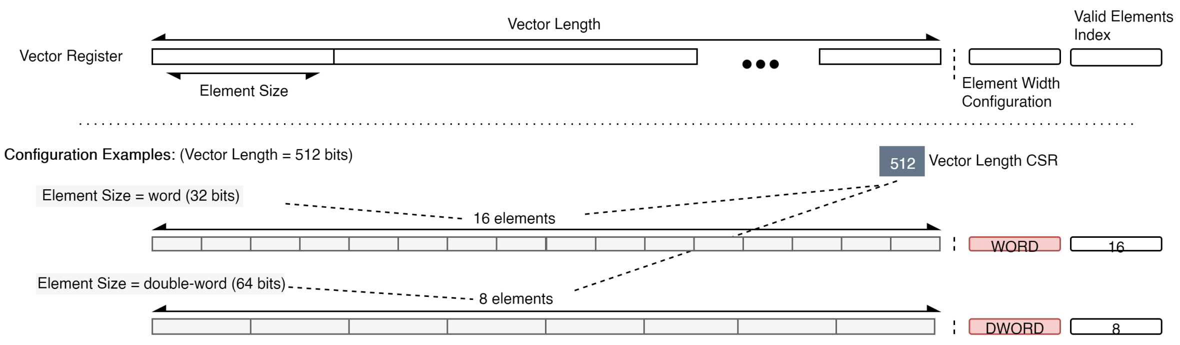

Vector Length
Valid Elements Index
Vector Register
Element Size
Element Width Configuration
Configuration Examples: (Vector Length = 512 bits)
512
Vector Length CSR
Element Size = word (32 bits)
16 elements
WORD
16
Element Size = double-word (64 bits)
8 elements
DWORD
8

Figure 2.18: Description of the structure of UVE vectorial register. On the bottom, two examples using different elements width and same vector length by Domingos et al. [2].

with predication. The first example is the vertical add instruction add, presented in Figure 2.19. This instruction takes as input two scalable vectors and performs the addition between them, saving the result on the destination register. As state before, the instruction must specify the data-type and, if the destination register is configured as a store streaming register, when writing to that register the data will be implicitly saved to memory by the streaming engine. The data-types supported are signed, unsigned and floating-point, and can be conjugated with all element widths, except for byte floating-point and half-word floating-point.

Another similar instruction to the vertical add is the horizontal add instruction $adde$, that takes as input one scalable vector and performs the summation of its elements, returning the result into a scalar register of the same element size. Both of these instruction also take as input a predication register, to specify which lanes should execute, all if vector ”p0” is used.

When it comes to logical instructions, the operations can be done using scalable vectors or scalar values. As an example, UVE implements a Shift Logical Left (SLL) for scalable vector with two versions available. Simple SLL takes as input two scalable vectors and uses the second one two perform individual shifts on the first vector. The result is saved into a scalable vector. The second version of this instruction is SLLS, where the last ”S” stands for scalar. As the name suggests, this operations takes only one scalable vector as input instead of two, and replaces the second with a scalar register. This instruction always performs the same amount of shifts on every element, unlike the previous one. Both instructions also make use of predication.

Another set of instructions featured in UVE are data transfer instructions. This are likely not going to be used as frequently because they go against one of the main purposes of streams and a decoupled streaming unit in this extension, to avoid memory instructions in the main processing unit. However, they may still be necessary in some application, so UVE implements a load instruction and a store instruction, each with 2 versions. The load/store instructions take as input the base memory from which memory is going to be read/stored and the number of elements that are going to be processed. The load instruction also requires specification about the elements width, on the instruction itself, while the store instruction does not need it because this information can be attained from the vector CSR. The reason for the 2

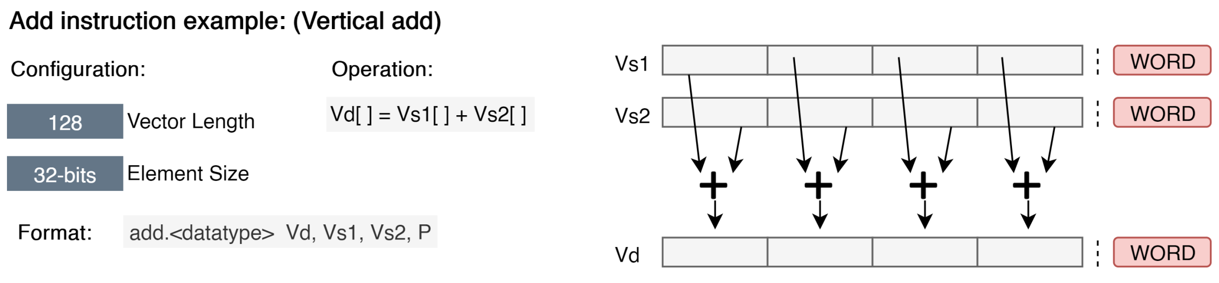


Figure 2.19: Description of UVE Add instruction by Domingos et al. [2].

types of instruction is to allow for the program to keep a coherent state even if the number of elements requested for read/load are not processed, because the size of the vector does not allow it. This way, the instruction updates the base memory or the size with the number of elements processed. Therefore, the instructions available are LOAD/STORE_A and LOAD/STORE_S, updating the address or the size, respectively. If the size requested is lower than the vector size, the number of valid elements is update on the vector configuration, making invalid the remaining elements. An example that illustrates the use of LOAD_A is presented in Figure 2.20.

To move data between registers UVE offers instructions to do scalar to/from vector transactions, vector to vector transactions and width conversions. When moving a vector to a scalar register, only the first element is moved and to a 64-bit register, because that's the size of the scalar registers. Going from a scalar register to a vector, the instruction needs a element width specification and the data from the scalar register is moved into the first element of the register, with possible loss of data. The is also a duplicating instruction that allows to replicate the data to all the vector elements. The instructions to move data from a register to another register do just that, with one of the variants allowing to transpose all the elements. This instructions also support predication.

Lastly, the width conversion instructions. To convert from a vector with smaller element width to a vector with a higher vector width, the data is copied from the left part of the source register into the destination register equal to the number of elements that can fit into the destination register. Then, the source register is shifted that many elements into the left, invalidating the elements on the right or loading new ones from the stream, if available. The data on the destination register can then be used inside a loop, as an example. Figure 2.21 illustrates how this instruction works. Width narrowing is also supported, with the inverse procedure from the previous instruction. It starts with a vector whose elements have a higher element width than the destination register, so it shifts the data on the destination register to the right an amount equal to the elements on the source register, and then copies that elements.

To use all the instructions before, it is necessary to first configure the streams. UVE has available

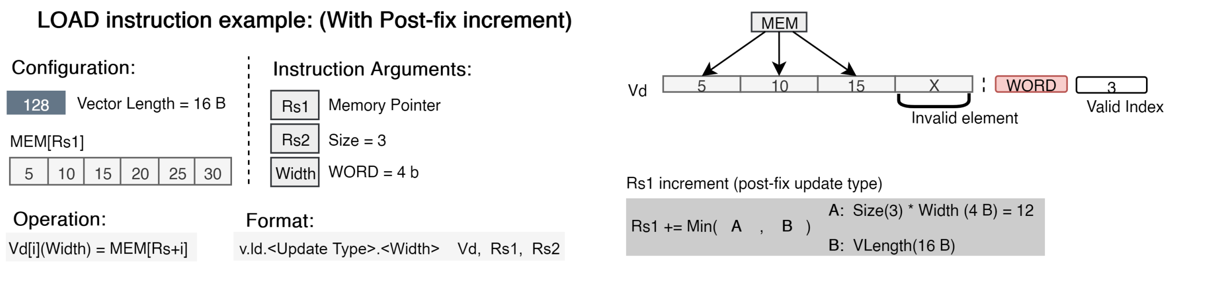


Figure 2.20: Example of the LOAD_A instruction by Domingos et al. [2]. The load instruction is configured to read 3 elements of 32-bit (word) size but the register is 16 bytes, so the last element of the vector is identified as an invalid element.

instructions to set up streams according to the principles described on Section 2.3.2. To configure a basic stream, the following instruction is used

ss.sta.<Dir>.<Width>   Vd, Rs1 (Size), Rs2 (Offset), Rs3 (Stride)

resulting in a one dimension descriptor of a stream. The ”Dir” and ”Width” represent if this is a loading/storing stream and the width of the elements, respectively. The three scalars on the instruction define the dimension size, base address (Offset) and the stride. The stream is associated with the vector in ”Vd”. When using only one dimension, the ”sta” in the instruction can be dropped. To associate more dimension to this vector register, UVE offers pretty much the same instruction but with the ”app” to append or ”end” to end the configuration of the stream in place of ”sta.<Dir>.<Width>”. The same register of destination must be used to configure the same stream with multiple dimensions in this instructions.

To represent more complex memory patterns, modifiers are used. There are two types of modifiers available: static and dynamic modifiers. Static modifiers are configured with the following disposition

ss.<type>.mod.<target>.<mode>   Vd, Rs1 (Size), Rs2 (Displacement)

where ”type” is one of ”app” to append to a dimension or ”end” to end a stream description. The others parameters were already detailed on Section 2.3.2 and the output stream ”Vd” must be a vector register whose dimension is to be associated with the modifier.

The indirect modifier is defined by the instruction

ss.<type>.ind.<target>.<mode>   Vd, Rs1 (Size), Vs2 (Source Stream)

that was also detailed on Section 2.3.2 and follows the same pattern as the previous one.

Lastly, for UVE to take advantage of streams during loops and remove the iterations control instructions, it presents custom branch instructions that depend on one of the dimensions of the stream being completed iterated or not. The instruction can be configured as

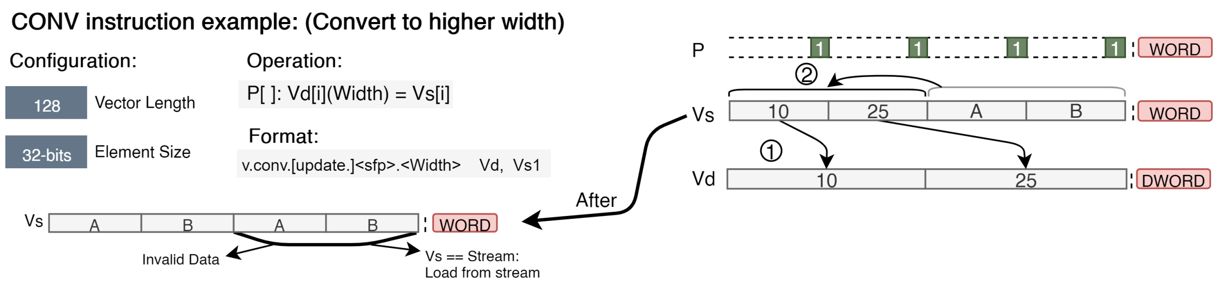

CONV instruction example: (Convert to higher width)
Configuration:
128
Vector Length
32-bits
Element Size
Operation:
P[ ]: Vd[i](Width) = Vs[i]
Format:
v.conv.[update.]<sfp>.<Width> Vd, Vs1
Vs
A
B
A
B
WORD
Invalid Data
Vs == Stream: Load from stream
After
P
1
1
1
1
WORD
②
Vs
10
25
A
B
WORD
①
Vd
10
25
DWORD

Figure 2.21: Conversion from a vector with word sized elements to a vector with double-word sized elements by Domingos et al. [2].

```
so.b.<condition>    Vd, <.gotolabel>
```

where the ”condition” can be ”c” for stream completed or ”nc” for stream not completed. This implementation only allows for the last dimension of a stream to be evaluated, but if intermediate dimensions need to be checked, the ”condition” changes from ”c” and ”nc” to ”dc.<dimension>” and ”ndc.<dimension>” respectively, where ”dimension” represents the vector dimension that the user wants to be checked during branch evaluation.

2.4 Related Work

Many works have already also made use of stream abstractions as a way to represent the flow of data, such as Imagine Stream Processor [20], RSVP [21], Q100 [22], Stream-dataflow acceleration [23], VEAL [24], and CoRAM++ [25]. However, none of these works target a general purpose out-of-order core and associated cache mechanisms. Another work that does not make use of stream abstractions is Memory Access Dataflow [26], a reconfigurable frontend/memory-fetch engine for accelerators and SIMD units, implemented as an in-core mechanism that is not part of the ISA.

The decoupling of memory processing from the main pipeline and execution of memory instructions physically separate as also been explored in other works. Outrider [27] enables the use of multiple simultaneous threads and provides memory latency tolerance to improve performance on highly threaded workloads. Outrider could outperform single threaded cores by 23-131% and a 4-way simultaneous multithreaded core by up to 87% on data parallel applications in a 1024-core system. Furthermore, Decoupled Supply-Compute (DeSC) [28] also decouples the main processing core from a second core or an accelerator that handles memory instructions. DeSC offers an average of 2.04$\times$ speedup over a baseline out-of-order single-chip multiprocessor and 1.56$\times$ speedup when a DeSC data supplier feeds data to a hardware accelerator.

When it comes to implementing streams on general purpose processors, Wang et al. [29] proposed an ISA extension for decoupled streams that can enable prefetch stream accesses and remove address computations instructions from the main core to hide some of the latency introduced by memory access operations. This work also implements compiler support using LLVM, that follows the process of first recognizing stream candidates and it’s selection and then generating the code for the target. Another work that handles streams with compiler support has also been developed by Neves et al. [30], where the frontend of LLVM is used to identify and modify memory accesses with a dedicated representation.

In regards to compiler implementations for new architectures and extensions, works [31] and [32] present LLVM extensions that support auto-vectorization. Some extensions can instead be implemented through directives, as in [33] and [34]. An accelerator that features compiler support, in specific in LLVM, is the Connex-S accelerator [35], that uses the LLVM framework and targets OPINCAA, a JIT vector assembler and coordination C++ library for Connex-S accelerating computations for an arbitrary CPU.

2.5 Summary

Throughout this chapter it was described the usual structure of a compiler, from starting to process the source code on the frontend, to the emission of the target machine code on the backend. To do so, compilers make use of intermediate representations, to help discover opportunities for optimizations, create independence from language and target specific code and to just simplify, for a more efficient execution of later stages.

In particular, the LLVM Infrastruture provides as a frontend for the class of C languages Clang. Clang makes use of LLVM IR for intermediate representation and the LLVM libraries to implement the backend. These libraries will be the main focus of this work on extending the LLVM Infrastructure to support a new scalabal vectorial extension.

Afterwards, it was discussed the principal extensions that are based on simple SIMD strategies, its advantages and disadvantages, and how scalable SIMD implementations try to solve those issues. However, the latter ones still leave room for improvement when it comes to memory access, by decoupling it from the processing unit, and by removing unnecessary instructions from the pipeline.

That is where the new experimental extension UVE comes in, by using a detached streaming unit and precise memory pattern descriptors, it can release the memory access from the main unit of execution and remove some instruction from loops.

3 Implementing UVE on LLVM

Contents

The main purpose of this work is to give compilation support for the new experimental extension UVE. The target to perform such extension is the LLVM Infrastructure, to take advantage of its assembly-like intermediate representation and its modular backend style, making it easier to implement such extension independent from the other components, outside of the backend target itself. The LLVM Infrastructure is also open source software that has been gaining traction and is already being used as a tool to give compiler support for Arm's SVE and RISC-V ”V” extensions [15–17].

UVE is implemented as an ISA extension to RISC-V, so the backend target is going be the group of libraries that handle this architecture. Throughout all the implementation, LLVM documentation will be used as reference [36–39].

3.1 LLVM Supporting Extension Requirements

To point out the main requirements for such implementation, it is necessary to dissect the differences from UVE to other extensions and architectures.

First, UVE makes use of scalable vectors, that differ from standard scalar registers, for arithmetic, logic and other operations but also to enable the use of predication, with it's own separate registers. One of the advantages of using UVE is that loop control instructions, such as incrementing the loop counter, are not necessary, because this control is done implicitly by the streaming engine. UVE also makes use of streams, and it's configuration, for multiple dimension and modifiers, is done by using the same register as the output. Finally, it has it's own instructions to execute operations between scalable vectors. With that, the following list of requirements is established:

1. The supporting implementation must be able to represent the scalable vectors and corresponding predication vectors with their own type and associated registers.
2. Remove the standard loop instructions, including the loop iteration control and the branch instruction itself, by replacing them with specific UVE loop control instructions.
3. Allow writing multiple times into the same same register to exploit UVE streaming mechanic, for configuration of streams and writing to a register associated with a stream, that performs an implicit store on the streaming engine.
4. Implement the rest of the extension instructions, following the template described on Section 2.3.3.
5. At the end, the supporting implementation must be able to fully translate an LLVM IR file written using all the elements described on the points stated above.

This list will be used as a reference throughout the implementation.

3.2 LLVM Support Implementation

In this section, it will be detailed all the steps taken to implement the backend support on LLVM for the experimental extension UVE. The objective is to fulfill all the requirements detailed in the previous section. Throughout this section, some code snipped may appear, to help show and explain the necessary changes for the implementation along with some examples. All the directories that will be mentioned from now forward will be relative to the LLVM base project root.

3.2.1 Setup

To start with, LLVM needs to know there is a new extension. UVE is an extension of the already existing architecture RISC V. In LLVM, such new extension is called a *SubTarget*, in this case, for RISC V.

For defining the existence of a new extension, it should be defined in `llvm/lib/Target/RISCV/RISCV.td` a new *SubtargetFeature*. The *SubtargetFeature* class manages the enabling and disabling of subtarget specific features [40]. Along with this class, it should also be defined a *Predicate* to tag elements that belong to this extension, such as instructions. Listing 3.1 shows the implementation of this classes for UVE.

Listing 3.1: Implementation of *SubtargetFeature* and *Predicate* classes.

```
def FeatureCustomExtUVE
    : SubtargetFeature<"experimental-xuve", "HasCustomExtUVE", "true",
                       "'UVE' (Unlimited Vector Extension Instructions)",
                       [FeatureStdExtF]>; // Needs Floating point
def HasCustomExtUVE
    : Predicate<"Subtarget->HasCustomExtUVE()">,
                       AssemblerPredicate<(all_of FeatureCustomExtUVE
    ),
                       "'UVE' (Unlimited Vector Extension
    Instructions)">;
```

Additionally, the information specified in Listing 3.1 should be added to `llvm/lib/Target/RISCV/RISCVSubtarget.h` as in Listing 3.2

Listing 3.2: Adding previously defined features to *RISCVSubtarget* class.

```
class RISCVSubtarget : public RISCVGenSubtargetInfo {
  virtual void anchor();
  bool HasStdExtM = false;
  bool HasStdExtA = false;
  ...
  bool HasCustomExtUVE = false;
  bool HasRV64 = false;
  bool IsRV32E = false;
  ...
  bool hasStdExtZbs() const { return HasStdExtZbs; }
  bool hasStdExtZbt() const { return HasStdExtZbt; }
  bool hasStdExtZbproposedc() const { return HasStdExtZbproposedc; }
  bool hasStdExtV() const { return HasStdExtV; }
  bool hasCustomExtUVE() const { return HasCustomExtUVE; }
  bool is64Bit() const { return HasRV64; }
  bool isRV32E() const { return IsRV32E; }
  ...
}
```

3.2.2 Register Definition

Now that the new extension is known by LLVM, it is necessary to define the registers supported by UVE. However, to define registers, it is necessary to specify the data type they will support. UVE uses scalable registers and scalable types. Such data types have already been defined before by Arm's SVE extension [41]. SVE represents the new Machine Value Types (MVT) for scalable vectors using the following structure:

nxv<size><type>

where <size> represents the minimum number of elements that the vector supports and <type> represents the basic type of the elements inside the scalable vector.

This register definition is done in `llvm/lib/Target/RISCV/RISCVRegisterInfo.td` as shown in Listing 3.3.

Listing 3.3: Defining UVE registers.

```
 1 // UVE registers
 2 let RegAltNameIndices = [ABIRegAltName] in {
 3   def U0   : RISCVReg<0, "u0", ["u0"]>;
 4   def U1   : RISCVReg<1, "u1", ["u1"]>;
 5   ...
 6   def U31  : RISCVReg<31,"u31", ["u31"]>;
 7 }
 8
 9 def UR : RegisterClass<"RISCV", [nxv8i8, nxv4i16, nxv2i32, nxv1i64], 64,
                                  (sequence "U%u", 0, 31)>;
10
11 // UVE registers
12 let RegAltNameIndices = [ABIRegAltName] in {
13   def P0   : RISCVReg<0, "p0", ["p0"]>;
14   ...
15   def P15  : RISCVReg<15,"p15", ["p15"]>;
16 }
17
18 class UPRClass<int lastreg> : RegisterClass< "RISCV",
19                               [ nxv16i1, nxv8i1, nxv4i1, nxv2i1 ], 16,
20                               (sequence "P%u", 0, lastreg)>{
21   let Size = 16;
22 }
23
24 def UPR     : UPRClass<15>;
25 def UPR_3b  : UPRClass<7>;
```

By analysing Listing 3.3 it is possible to see that first the register are defined by instantiating the class *RISCVReg* and then a class of registers is defined containing all the data types that it supports, by instantiating the class *RegisterClass*. The *UR* class of registers is defined as having 32 scalable registers, from 0 to 31, to mirror the architecture discussed previously and supports all the MVT described in line 9. For the predicate registers, two class of registers are defined: *UPR* and *UPR_3b*. The reason for two separate definitions is that, like mentioned previously, UVE has 16 separate scalable predicate registers but only 8 can perform arithmetic operations, in specific the register from 0 to 7.

Both of this classes supported MVT are calculated to match the minimum vector length, 64 bits for the scalable registers and 16 bits for the scalable predicate registers.

3.2.3 UVE Instructions Definition

Now that the extensions registers are defined, its time to define the target instructions. This is usually done by instantiating the instruction class from the target. Since this extension targets RISC V, the class that is going to be instantiated is *RVInst*, that is defined in Listing 3.4

Listing 3.4: Definition of *RVInst* class

```
1  class RVInst<dag outs, dag ins, string opcodestr, string argstr,
2               list<dag> pattern, InstFormat format>
3      : Instruction {
4    field bits<32> Inst;
5    field bits<32> SoftFail = 0;
6    let Size = 4;
7
8    bits<7> Opcode = 0;
9
10   let Inst{6-0} = Opcode;
11
12   let Namespace = "RISCV";
13
14   dag OutOperandList = outs;
15   dag InOperandList = ins;
16   let AsmString = opcodestr # "\t" # argstr;
17   let Pattern = pattern;
18
19   let TSFlags{4-0} = format.Value;
20
21   // Defaults
22   RISCVVConstraint RVVConstraint = NoConstraint;
23   let TSFlags{8-5} = RVVConstraint.Value;
24 }
```

where the arguments ”outs” represents the instruction output, ”ins” represents the instruction inputs, ”opcodestr” is the string that is going to be written in assembly for this instruction's operation code (ex. ss.sta.ld.w), ”argstr” is the string that will be written to assembly as this instruction's arguments, ”pattern” can be used to associate a pattern to this instruction and ”format” is a flag representing the instruction format. This class defines an instruction as having 32 bits, as expected, in line 4.

Two files are used to define the new instructions: the file responsible to keep RISC V instruction formats, `llvm/lib/Target/RISCV/RISCVInstrFormatsUVE.td`, and the file responsible to keep RISC V instruction info, `llvm/lib/Target/RISCV/RISCVInstrInfoUVE.td`.

The RISC V instruction formats used during this work are going to be 2: the R-type format and the B-type format. The R-type is used to encode instructions that do operations between registers, such as arithmetic operations, and is the most basic type. The B-type format is used to encode conditional branch instructions. The format of both of these instructions is shown on Table 3.1.

Following the order used in Section 2.3.3.B, the first instructions to be implemented are the arithmetic instructions.

3.2.3.A Implementing UVE Arithmetic Instructions

On `llvm/lib/Target/RISCV/RISCVInstrFormatsUVE.td`, start by defining a class that will represent arithmetic instructions with one scalable vector as output and two scalable vector and 1 scalable predicate register as input. This will be a derived class that will inherit all the attributes from the class *RVInst*,

Table 3.1: RISC V R-type and B-type formats, respectively from top to bottom [3].

RISC-V encoding types								
bits(31:25)		bits(24:20)	bits(19:15)	bits(14:12)	bits(11:7)		bits(6:0)
funct7		RS2	RS1	funct3	RD		opcode
imm[12]	imm[10:5]	RS2	RS1	funct3	imm[4:1]	imm[11]	opcode

Table 3.2: UVE Arithmetic-type format.

Arithmetic Instruction Type									
bits(31:25)			bits(24:20)	bits(19:15)	bits(14:12)			bits(11:7)	bits(6:0)
Arithmetic Opc.	dec(2:1)	Predicate	RS2	RS1	dec(0)	s	FP	RD	StreamOps

shown on Listing 3.4, and can be seen on Listing 3.5.

Listing 3.5: Definition of *UVE_ARITH_V_VVP_f* derived class

```
1 class UVE_ARITH_V_VVP_f<bits<4> funct4, bits<1> funct1, UVE_DataType
     usftype,
2                         RISCVOpcode opcode, dag outs, dag ins, string
     opcodestr, string argstr>
3     : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
4   bits<3> ps1;
5   bits<5> us2;
6   bits<5> us1;
7   bits<5> ud;
8
9   let Inst{31-28} = funct4;
10  let Inst{27-25} = ps1;
11  let Inst{24-20} = us2;
12  let Inst{19-15} = us1;
13  let Inst{14}    = funct1;
14  let Inst{13-12} = usftype.Value;
15  let Inst{11-7}  = ud;
16  let Opcode = opcode.Value;
17 }
```

This derived class uses *InstFormatR* on line 3 to define this instruction as a RISC V R-type instruction and then proceeds to place the input variables into the respective bit fields. Setting the attributes follows the format specified on Table 3.2, that is a UVE adaptation of the R-type format.

The derived class in Listing 3.5 is used as a template for all the arithmetic instructions that have as output one scalable vector and as input two scalable vectors and one scalable predicate vector, therefore the name *ARITH_V_VVP*.

To implement the vertical add instruction, it is first necessary to define multiple classes that instantiate from *UVE_ARITH_V_VVP_f*. LLVM TableGen tool has a feature called *multiclasses* that makes this job easier. By defining a variable associated with a multiclass, TableGen will generate all the possible combinations inside it. To better explain it, the implementation of the class and multiclass used to define the vertical add instruction is in Listing 3.6.

Listing 3.6: Definition of class and multiclass to define arithmetic instructions.

```
1  // mayLoad = 1 because vector ALU ops read the configuration CSRs
2  // and we model that as memory operand.
3  let hasSideEffects = 0, mayLoad = 1, mayStore = 1 in {
4    // UVE Arithmetic-V-VVP Classes
5    class UVE_ARITH_V_VVP<bits<4> funct4, bits<1> funct1, UVE_DataType
       usftype, string opcodestr>
6          : UVE_ARITH_V_VVP_f<funct4, funct1, usftype, OPC_UVE_OP,
7                    (outs UR:$ud), (ins UR:$us1, UR:$us2, UPR_3b:$ps1),
8                    opcodestr, "$ud, $us1, $us2, $ps1">;
9
10   multiclass UVE_ARITH_V_VVP_USF<bits<4> funct4, bits<1> funct1, string
       opcodestr>{
11      def _U        : UVE_ARITH_V_VVP<funct4, funct1, UVE_DataType_Unsigned
        , opcodestr # ".us">;
12      def _S        : UVE_ARITH_V_VVP<funct4, funct1, UVE_DataType_Signed
        , opcodestr # ".sg">;
13      def _FP       : UVE_ARITH_V_VVP<funct4, funct1,
       UVE_DataType_FloatingP, opcodestr # ".fp">;
14   }
15   ...
16 }
```

In Listing 3.6 a first class is instantiated from the one in Listing 3.5, and itself is instantiated multiple times by the multiclass. Then, when defining a variable from this multiclass, multiple variables will be defined, each one with the strings after the *def* keyword (_U, _S and _FP) appended to the name of the variable. This way it is only necessary to use one definition and multiple instructions will be defined from the multiclass. The flags set in line 3 are used to inform the rest of the compiler about the instruction memory access and side effects, to decide if it can perform certain optimizations or not, such as Dead Code Elimination (DCE).

So now, by using the *defm* keyword (because it's instantiating from a multiclass, not a class)

Listing 3.7: Definition of arithmetic instructions.

```
let Predicates = [HasCustomExtUVE] in {

defm   UVE_ADD    : UVE_ARITH_V_VVP_USF<0b0000, 0b0, "so.a.add">;
defm UVE_SUB    : UVE_ARITH_V_VVP_USF<0b0000, 0b1, "so.a.sub">;
defm UVE_MUL    : UVE_ARITH_V_VVP_USF<0b0001, 0b0, "so.a.mul">;
defm UVE_DIV    : UVE_ARITH_V_VVP_USF<0b0001, 0b1, "so.a.div">;
defm UVE_MAC    : UVE_ARITH_V_VVP_USF<0b0011, 0b1, "so.a.mac">;
defm UVE_ABS    : UVE_ARITH_V_VVP_USF<0b0011, 0b0, "so.a.abs">;

defm UVE_MIN    : UVE_ARITH_V_VVP_USF<0b0100, 0b0, "so.a.min">;
defm UVE_MAX    : UVE_ARITH_V_VVP_USF<0b0100, 0b1, "so.a.max">;

defm UVE_MINE   : UVE_ARITH_V_VP_USF<0b0101, 0b0, "so.a.mine">;
defm UVE_MAXE   : UVE_ARITH_V_VP_USF<0b0101, 0b1, "so.a.maxe">;

defm UVE_DEC    : UVE_ARITH_V_VP_USF<0b0110, 0b0, "so.a.inc">;
defm UVE_INC    : UVE_ARITH_V_VP_USF<0b0110, 0b1, "so.a.dec">;

defm UVE_ADDE   : UVE_ARITH_V_VP_ACC_USF_ACC<0b0010, 0b0, "so.a.adde">;
defm UVE_ADDS   : UVE_ARITH_S_VP_ACC_USF_ACC<0b0010, 0b1, "so.a.adds">;
defm UVE_ADDF   : UVE_ARITH_F_VP_ACC_USF_ACC<0b0010, 0b1, "so.a.adds">;


} // Predicates = [HasCustomExtUVE]
```

after the class and multiclass definition, TableGen will create three vertical add instructions, for unsigned, signed and floating-point operators, with variable names *UVE_ADD_U*, *UVE_ADD_S* and *UVE_ADD_FP*, respectively, as presented in Listing 3.7 highlighted. The same can be done for the all the rest of the arithmetic instructions, by using the same multiclass or by creating another using the same method that matches the inputs and outputs of each one and their register class, setting the at-

Table 3.3: UVE Logic-type format.

Logic Instruction Type							
bits(31:25)			bits(24:20)	bits(19:15)	bits(14:12)	bits(11:7)	bits(6:0)
Logical Opcode	dec(3)	Predicate	RS2	RS1	dec(2:0)	RD	StreamOps

tributes correctly. By inspecting the code in Listing 3.6, it is possible to deduce the outputs and inputs of each instruction, where ”V” stands for scalable register, ”S” for scalar register and ”P” for scalable predicate register.

3.2.3.B Implementing UVE Logic Instructions

For UVE logic instructions, the instruction format used changes to the one on Table 3.3.

The procedure behind supporting UVE logic instructions on the backend follows the same done for the arithmetic instructions. First, a class is instantiated from *RVInst* functioning as a template for UVE logic instructions in `llvm/lib/Target/RISCV/RISCVInstrFormatsUVE.td`, such as in Listing 3.8.

Listing 3.8: Definition of *UVE_LOGIC_V_VSP_f* derived class.

```
class UVE_LOGIC_V_VSP_f <bits<4> funct4, bits<3> funct3,
                     RISCVOpcode opcode, dag outs, dag ins, string
   opcodestr, string argstr>
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
  bits<3> ps1;
  bits<5> rs2;
  bits<5> us1;
  bits<5> ud;

  let Inst{31-28} = funct4;
  let Inst{27-25} = ps1;
  let Inst{24-20} = rs2;
  let Inst{19-15} = us1;
  let Inst{14-12} = funct3;
  let Inst{11-7}  = ud;
  let Opcode = opcode.Value;
}
```

Second, derive more specific classes in `llvm/lib/Target/RISCV/RISCVInstrInfoUVE.td` from the previous class, such as in Listing 3.9.

Listing 3.9: Definition of class to define logic instructions.

```
// mayLoad = 1 because vector ALU ops read the configuration CSRs
// and we model that as memory operand.
let hasSideEffects = 0, mayLoad = 1, mayStore = 1 in {
  ...
  // UVE Logic-V-VSP Classes
  class UVE_LOGIC_V_VSP<bits<4> funct4, bits<3> funct3, string opcodestr>
        : UVE_LOGIC_V_VSP_f<funct4, funct3, OPC_UVE_OP,
                (outs UR:$ud), (ins UR:$us1, GPR:$rs2, UPR_3b:$ps1),
                opcodestr, "$ud, $us1, $rs2, $ps1">;
}
```

Finally, define the logic instruction by instantiating from the more specific derived classes, as done in Listing 3.10.

Table 3.4: UVE Branch-type format.

Loop Control Instruction Types									
bits(31:25)			bits(24:20)	bits(19:15)	bits(14:12)		bits(11:7)		bits(6:0)
Loop Opcode	imm[12]	x	imm[10:5]	RS1	0	dec(1:0)	imm[4:1]	imm[11]	StreamOps
Loop Opcode	imm[12]	imm[10:5]	Predicate2	Predicate	1	dec(1:0)	imm[4:1]	imm[11]	StreamOps

Listing 3.10: Definition of logic instructions.

```
let Predicates = [HasCustomExtUVE] in {

def UVE_NAND  : UVE_LOGIC_V_VVP<0b1100, 0b000, "so.a.nand">;
def UVE_AND   : UVE_LOGIC_V_VVP<0b1100, 0b001, "so.a.and">;
def UVE_NOR   : UVE_LOGIC_V_VVP<0b1100, 0b010, "so.a.nor">;
def UVE_OR    : UVE_LOGIC_V_VVP<0b1100, 0b011, "so.a.or">;
def UVE_XOR   : UVE_LOGIC_V_VVP<0b1100, 0b101, "so.a.xor">;
def UVE_SLL   : UVE_LOGIC_V_VVP<0b1101, 0b000, "so.a.sll">;
def UVE_SRL   : UVE_LOGIC_V_VVP<0b1101, 0b010, "so.a.srl">;
def UVE_SRA   : UVE_LOGIC_V_VVP<0b1101, 0b100, "so.a.sra">;

def  UVE_SLLS  : UVE_LOGIC_V_VSP<0b1101, 0b001, "so.a.slls">;
def UVE_SRLS  : UVE_LOGIC_V_VSP<0b1101, 0b011, "so.a.srls">;
def UVE_SRAS  : UVE_LOGIC_V_VSP<0b1101, 0b101, "so.a.sras">;

def UNOT   : UVE_LOGIC_V_VP<0b1100, 0b100, "so.a.not">;
} // Predicates = [HasCustomExtUVE]
```

The procedure to implement the rest of the instructions is the same as the one for the arithmetic and logic instructions, only changing the instruction format. However, there is one group of instructions that has a difference: the branch instructions.

3.2.3.C Implementing UVE Branch Instructions

Instead of using the RISC V R-type format for instruction encoding, branch instructions use an adaptation of the B-type format, presented on Table 3.4.

To create the base class, the procedure is the same: instantiate from *RVInst* and set the attributes according to the format on Table 3.4.

Listing 3.11: Definition of *UVE_BRANCH_f* derived class.

```
1  class UVE_BRANCH_f<UVE_BranchComplete comp, UVE_StreamHop hop,
2                     RISCVOpcode opcode, dag outs, dag ins, string
     opcodestr, string argstr>
3    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatUVESB> {
4    bits<12> imm12;
5    bits<5> compat;
6    bits<5> us1;
7
8    let Inst{31-29} = 0b111;
9    let Inst{28}    = imm12{11};
10   let Inst{27-22} = imm12{9-4};
11   let Inst{21}    = comp.Value;
12   let Inst{20}    = hop.Value{2};
13   let Inst{19-15} = us1;
14   let Inst{14-13} = hop.Value{1-0};
15   let Inst{12}    = 0;
16   let Inst{11-8}  = imm12{3-0};
17   let Inst{7}     = imm12{10};
18   let Opcode      = opcode.Value;
19 }
```

However this time, there is a difference on the flags set, as highlighted on line 3 in Listing 3.11.

This particular flag was defined before, with the objective of using it on branch instructions to emit a fixup. In LLVM, fixups are used to represent information in instructions which is currently unknown (such as a memory location of an external symbol). During instruction encoding the unknown information is encoded as if the value is equal to 0 and a fixup is emitted which contains information on how to rewrite the value when information is known [42]. If the fixup can not be resolved before the ELF, it is converted into a relocation.

So, in this case, this flag will emit a *fixup_riscv_uve_stream_branch*. Then, on `llvm/lib/Target/RISCV/MCTargetDesc/RISCVAsmBackend.cpp` the backend will adjust the fixup value, accordingly to Listing 3.12.

Listing 3.12: Fixup value adjustment.

```
static uint64_t adjustFixupValue(const MCFixup &Fixup, uint64_t Value,
                                 MCContext &Ctx) {
  switch (Fixup.getTargetKind()) {
  default:
    llvm_unreachable("Unknown fixup kind!");
  ...
  case RISCV::fixup_riscv_uve_stream_branch: {
    if (!isInt<13>(Value))
      Ctx.reportError(Fixup.getLoc(), "fixup value out of range");
    if (Value & 0x1)
      Ctx.reportError(Fixup.getLoc(), "fixup value must be 2-byte aligned");
    // Need to extract imm[12], imm[10:5], imm[4:1], imm[11] from the 13-bit
    // Value.
    unsigned Sbit = (Value >> 12) & 0x1;
    unsigned Hi1 = (Value >> 11) & 0x1;
    unsigned Mid6 = (Value >> 5) & 0x3f;
    unsigned Lo4 = (Value >> 1) & 0xf;
    // Inst{28} = Sbit;
    // Inst{27-22} = Mid6;
    // Inst{11-8} = Lo4;
    // Inst{7} = Hi1;
    Value = (Sbit << 28) | (Mid6 << 22) | (Lo4 << 8) | (Hi1 << 7);
    return Value;
  }
  ...
  }
}
```

Latter on, the assembler will be able to use this value to apply a fixup and place the correct memory address that the branch will have to jump to. If not able to do so, it will emit a relocation.

After that, the procedure is to define the branch instructions is the same as the other instructions, first defining a branch instruction class template by instantiating the class in Listing 3.11 and then defining an instruction from that class.

3.2.4 Creating support for UVE instructions in LLVM IR

Now that UVE instructions are defined on the backend, it is time to use them. To do this, there are two possible ways:

- **LLVM Intrinsics**

  LLVM instrinsics are similar to function calls in C language. They are defined by declaring its inputs, outputs and optionally any flags and they are transparent to optimization passes.

- **LLVM Instructions**

  Instructions, in the LLVM IR sense, are more similar to assembly instructions, but because they are used in LLVM IR they still have a return value before the instruction. They are not transparent to optimization passes.

When considering using an instruction, the main advantage is being able to use optimization passes for any manipulation or optimizations that may seem necessary. However, when compared to intrinsics, this are much more intricate. Because UVE uses streams, optimizations become more complex and may not even be supported for this kind of representation. Also, LLVM suggests that, if the intended functionality can be represented as a function call, then it should probably start as an intrinsic. Therefore, the method this work uses to implement a representation of UVE instructions in LLVM IR are the intrinsics.

3.2.4.A Defining LLVM Instrinsics

To start defining new instruction it is necessary to instantiate them from the *Intrinsic* class, defined in Listing 3.13.

Listing 3.13: Definition of *Intrinsic* class.

```
class Intrinsic<list<LLVMType> ret_types,
                list<LLVMType> param_types = [],
                list<IntrinsicProperty> intr_properties = [],
                string name = "",
                list<SDNodeProperty> sd_properties = []> :
    SDPatternOperator {
  string LLVMName = name;
  string TargetPrefix = "";   // Set to a prefix for target-specific
    intrinsics.
  list<LLVMType> RetTypes = ret_types;
  list<LLVMType> ParamTypes = param_types;
  list<IntrinsicProperty> IntrProperties = intr_properties;
  let Properties = sd_properties;

  bit isTarget = 0;
}
```

This class takes as first argument a list of return types that the intrinsic will support, the second argument takes a list of input types and the third argument takes a list of intrinsic properties.

Before starting to define new intrinsics, it is useful to separate UVE intrinsics from the others given the structure of the compiler being modular, so they are in `llvm/include/llvm/IR/IntrinsicsRISCVUVE.td`

To implement arithmetic intrinsics it will be used the same methodology as for defining new instructions: derive a new class from the *Intrinsic* class and then define the intrinsic by instantiating that class or others multiclasses.

Listing 3.14: Definition of arithmetic derived classes and arithmetic intrinsics.

```
let TargetPrefix = "riscv" in {

//Intrinsics for UVE Arithmetic and MISC Instructions

  class INTRINSIC_UVE_ARITH_V_VVP<LLVMType itype>
        : Intrinsic<[itype], [LLVMMatchType<0>, LLVMMatchType<0>, itype],
      [IntrWriteMem]>;

  multiclass INTRINSIC_UVE_ARITH_V_VVP_USF{
    def _u        : INTRINSIC_UVE_ARITH_V_VVP<llvm_anyvector_ty>;
    def _s        : INTRINSIC_UVE_ARITH_V_VVP<llvm_anyvector_ty>;
    def _fp       : INTRINSIC_UVE_ARITH_V_VVP<llvm_anyvector_ty>;
  }

  defm int_riscv_uve_add : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_sub : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_mul : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_div : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_mac : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_abs : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_min : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_max : INTRINSIC_UVE_ARITH_V_VVP_USF;
  ...

} // TargetPrefix = "riscv"
```

In Listing 3.14, the class *INTRINSIC_UVE_ARITH_V_VVP* is defined as being an intrinsic that has one return type and three input parameters, with the first two input parameters needing to match the return type. Then, a multiclass is instantiated from the previous class, passing as parameter the type *llvm_anyvector_ty*. This makes it so the return type of the intrinsic can be any of the scalable vector types, while the two first parameters must match the return type. The third and last parameter can also be of any vector type, to define it later as one of the scalable predication vector types. This definition matches the pattern of the arithmetic instructions defined previously. Lastly, all the intrinsics that share this pattern are defined using such classes.

This procedure is replicated for all the other intrinsics to mimic the corresponding UVE instructions defined in the backend, with the exception of the branch intrinsics.

The branch intrinsics, instead of following the same pattern of the UVE branch instructions, will be used as a part of the pattern for the instruction selection, and not as a whole. The reason is that LLVM IR requires a terminating instruction at the end of a block. As this is an intrinsic, it will be used in conjuntion with a branch instruction to represent the same functionality.

As presented in Listing 3.15, the branch intrinsics are defined as having one output of integer type and one input of any vector type, that will be a scalable vector. The output of this intrinsic will be used in LLVM IR code as input to one conditional branch instruction, that will later during the instruction selection step be all converted into UVE's corresponding branch instruction.

3.2.4.B Creating Patterns

With the UVE instructions and LLVM intrinsics already defined, it is necessary to make a connection between them, so that, during instruction selection, the compiler knows what UVE instruction to select to replace the LLVM intrinsic. This association is done through patterns.

Listing 3.15: Definition of branch derived classes and branch intrinsics.

```
// Intrinsics for UVE Branch

  class INTRINSIC_UVE_BRANCH
      : Intrinsic<[llvm_anyint_ty], [llvm_anyvector_ty], [IntrNoMem]>;

  multiclass INTRINSIC_UVE_BRANCH_HOP{
    def _1      : INTRINSIC_UVE_BRANCH;
    def _2      : INTRINSIC_UVE_BRANCH;
    def _3      : INTRINSIC_UVE_BRANCH;
    def _4      : INTRINSIC_UVE_BRANCH;
    def _5      : INTRINSIC_UVE_BRANCH;
    def _6      : INTRINSIC_UVE_BRANCH;
    def _7      : INTRINSIC_UVE_BRANCH;
    def _1      : INTRINSIC_UVE_BRANCH;
  }

  multiclass INTRINSIC_UVE_BRANCH_HOP_COMPLETE{
    defm _comp  : INTRINSIC_UVE_BRANCH_HOP;
    defm _ncmp  : INTRINSIC_UVE_BRANCH_HOP;
  }

  defm int_riscv_uve_branch : INTRINSIC_UVE_BRANCH_HOP_COMPLETE;
```

Patterns allow to associate backend instructions to a certain pattern that can occur in LLVM IR. The simplified pattern class *Pat* will take one pattern to associate with one instruction. Since the intrinsics were implement as a one to one match with UVE instructions, with the exception of the branch, the *Pat* class will be used.

During the intrinsic implementation the type ”*any*” was used, so it is necessary to specify in the pattern the types of the operators, as the compiler can throw errors if it can’t figure it out. Some of the logic intrinsic-instruction patterns are implemented in Listing 3.16. Each definition between an intrinsic and an instruction instantiates four times the *Pat* class, for different data types. This pattern will later be used during instruction selection to replace the intrinsic by the associated instruction inside the pattern.

Listing 3.16: Patterns between logic intrinsics and logic instructions.

```
//Template classes for some patterns
multiclass Pattern_UVE_V_VVP<Intrinsic intrinsic, Instruction instruction
    >{
  def : Pat<(nxv8i8 (intrinsic (nxv8i8 UR:$us1), (nxv8i8 UR:$us2), (
    nxv16i1 UPR_3b:$ps1))), (instruction UR:$us1, UR:$us2, UPR_3b:$ps1)>;
  def : Pat<(nxv4i16 (intrinsic (nxv4i16 UR:$us1), (nxv4i16 UR:$us2), (
    nxv8i1 UPR_3b:$ps1))), (instruction UR:$us1, UR:$us2, UPR_3b:$ps1)>;
  def : Pat<(nxv2i32 (intrinsic (nxv2i32 UR:$us1), (nxv2i32 UR:$us2), (
    nxv4i1 UPR_3b:$ps1))), (instruction UR:$us1, UR:$us2, UPR_3b:$ps1)>;
  def : Pat<(nxv1i64 (intrinsic (nxv1i64 UR:$us1), (nxv1i64 UR:$us2), (
    nxv2i1 UPR_3b:$ps1))), (instruction UR:$us1, UR:$us2, UPR_3b:$ps1)>;
}

//Patterns for UVE Logic Instructions-Intrinsics

defm : Pattern_UVE_V_VVP<int_riscv_uve_nand, UVE_NAND>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_and, UVE_AND>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_nor, UVE_NOR>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_or, UVE_OR>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_xor, UVE_XOR>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_sll, UVE_SLL>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_srl, UVE_SRL>;
defm : Pattern_UVE_V_VVP<int_riscv_uve_sra, UVE_SRA>;
```

Again, the odd one out are the patterns between the branch intrinsics and the branch instructions. As explained before, LLVM IR needs to have a terminating instruction at the end of a basic block. A solution to this problem is use the return value of the branch intrinsic as input to the conditional branch at the end of the block. This way, LLVM IR is well-formed and by recreating that same pattern it is possible to replace all with the UVE branch instruction. The pattern that needs to be used is on lines 4, 5 and 6 of Listing 3.17. This is the pattern that is reproduced by the LLVM backend at the time of instruction selection when processing the sequence of LLVM IR instructions and intrinsics described previously. The pattern takes the output of the branch intrinsic, performs an *and* with the return value and the value 1, and then uses the returning value as input to the *brcond* instruction, with second operand the loop target label.

Listing 3.17: Patterns between branch intrinsics and branch instructions.

```
1  //  Patterns for Branch Instructions-Intrinsics
2
3  multiclass Pattern_UVE_BRANCH<Intrinsic intrinsic, Instruction
       instruction>{
4    def : Pat<(brcond (and (intrinsic (nxv8i8 UR:$us1)), (i64 1)), bb:$loop
       ),              (instruction UR:$us1, simm13_lsb0:$loop)>;
5    def : Pat<(brcond (and (intrinsic (nxv4i16 UR:$us1)), (i64 1)), bb:
       $loop),              (instruction UR:$us1, simm13_lsb0:$loop)>;
6    def : Pat<(brcond (and (intrinsic (nxv2i32 UR:$us1)), (i64 1)), bb:
       $loop),              (instruction UR:$us1, simm13_lsb0:$loop)>;
7    def : Pat<(brcond (and (intrinsic (nxv1i64 UR:$us1)), (i64 1)), bb:
       $loop),              (instruction UR:$us1, simm13_lsb0:$loop)>;
8  }
9
10 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_1, UVE_BRANCH_COMP_1
       >;
11 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_2, UVE_BRANCH_COMP_2
       >;
12 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_3, UVE_BRANCH_COMP_3
       >;
13 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_4, UVE_BRANCH_COMP_4
       >;
14 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_5, UVE_BRANCH_COMP_5
       >;
15 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_6, UVE_BRANCH_COMP_6
       >;
16 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_7, UVE_BRANCH_COMP_7
       >;
17 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_comp_l, UVE_BRANCH_COMP_L
       >;
18 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_1, UVE_BRANCH_NCMP_1
       >;
19 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_2, UVE_BRANCH_NCMP_2
       >;
20 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_3, UVE_BRANCH_NCMP_3
       >;
21 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_4, UVE_BRANCH_NCMP_4
       >;
22 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_5, UVE_BRANCH_NCMP_5
       >;
23 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_6, UVE_BRANCH_NCMP_6
       >;
24 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_7, UVE_BRANCH_NCMP_7
       >;
25 defm : Pattern_UVE_BRANCH<int_riscv_uve_branch_ncmp_l, UVE_BRANCH_NCMP_L
       >;
```

After all that, LLVM backend should be able to take one file written in LLVM IR with UVE intrinsics and transform it into an assembly file with UVE instructions. However, one big problem remains that was not considered up until now: LLVM IR is in SSA form.

3.2.4.C Overcoming LLVM IR SSA Form

The problem posed by LLVM IR SSA form is that UVE makes use of storing streams by writing to them, and the streaming unit performs the necessary operations to save that data, implicitly. However, when writing LLVM IR it is not allowed to write to the same variable more than once. This poses a problem because the compiler will assume that the result of the instruction is not to be saved on the same register as the loading stream, writing the final assembly code with different scalable registers.

To overcome this problem, the first solution was to add an additional input to the instruction, with the extra input being the variable associated with the streaming register. Then, by setting constraints on the instruction, it was possible to force the output register to be the same as the extra input register. This two registers then become tied, and are treated in a special way on the backend. To remove two linking registers, the compiler during code emission will run a pass called *TwoAddressInstructionPass.cpp*. This pass has as function to solve such problems, and to do so it emits a *COPY* instruction before the instruction with tied registers and uses as input the tied register and as output a different register. The tied register in the instruction ahead is then replaced by this output register. The result would look something like Figure 3.1, with the implementation of the *COPY* instruction being target dependent.

This does not solve the problem at all.

The second solution makes use of pseudo-instruction. Pseudo-instructions are machine instruction that don't correspond to any specific assembly instruction. They are used as a placeholder to later be replaced by a pass. In this case, the pass will be *ExpandPostRAPseudos.cpp*.

So, the idea behind this type of solution is to create a pseudo-instruction that is similar in outputs to the corresponding UVE instruction but takes one additional input: the stream variable. This pseudo-instruction will later be replaced by the pass for the intended UVE instruction with the extra input register that was allocated as an output register for that instruction. To give it LLVM IR compatibility, a new intrinsic is also created for the instructions that may want to save the returning value into a streaming register. One peculiar case are the stream configuration instructions. Apart from the starting instruction, all others (appending extra dimensions, creating direct and indirect modifiers) need to use a streaming register as input, so in this case it will not be defined a new intrinsic but it will replace the previous one, as it is incorrect. By taking this changes into consideration, the code in Listing 3.14 now becomes the code in Listing 3.18, with the differences highlighted.

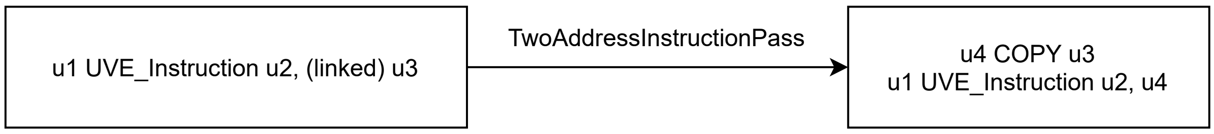

u1 UVE_Instruction u2, (linked) u3
TwoAddressInstructionPass
u4 COPY u3
u1 UVE_Instruction u2, u4

Figure 3.1: Example instruction before and after the two address instruction pass.

Listing 3.18: Definition of arithmetic derived classes and arithmetic intrinsics with pseudo-instructions in mind.

```
let TargetPrefix = "riscv" in {

//Intrinsics for UVE Arithmetic and MISC Instructions

  class INTRINSIC_UVE_ARITH_V_VVP<LLVMType itype>
        : Intrinsic<[itype], [LLVMMatchType<0>, LLVMMatchType<0>, itype],
      [IntrWriteMem]>;

  class INTRINSIC_UVE_ARITH_V_VVVP<LLVMType itype>
        : Intrinsic<[itype], [LLVMMatchType<0>, LLVMMatchType<0>,
        LLVMMatchType<0>, itype], [IntrWriteMem]>;

  multiclass INTRINSIC_UVE_ARITH_V_VVP_USF{
    def _u        : INTRINSIC_UVE_ARITH_V_VVP<llvm_anyvector_ty>;
    def _s        : INTRINSIC_UVE_ARITH_V_VVP<llvm_anyvector_ty>;
    def _fp       : INTRINSIC_UVE_ARITH_V_VVP<llvm_anyvector_ty>;
    def _u_save       : INTRINSIC_UVE_ARITH_V_VVVP<llvm_anyvector_ty>;
    def _s_save       : INTRINSIC_UVE_ARITH_V_VVVP<llvm_anyvector_ty>;
    def _fp_save     : INTRINSIC_UVE_ARITH_V_VVVP<llvm_anyvector_ty>;
  }

  defm int_riscv_uve_add : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_sub : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_mul : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_div : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_mac : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_abs : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_min : INTRINSIC_UVE_ARITH_V_VVP_USF;
  defm int_riscv_uve_max : INTRINSIC_UVE_ARITH_V_VVP_USF;
  ...
}
```

As stated before, the intrinsic will now take one extra argument, the target scalable register. Now arithmetic operations using scalable registers can now be performed in LLVM IR with the intention of saving the result into a stream (by using *_save* at the end) or not.

Still, the definition of the corresponding pseudo-instructions and patterns is necessary to generate the target code. pseudo-instructions share similarities with regular instructions in it's definition, but it uses the *Pseudo* class to derive a class, intead of *RVInst*. Listing 3.19 shows how some of the arithmetic pseudo-instructions are implemented.

Listing 3.19: Definition of pseudo arithmetic derived classes and arithmetic pseudo-instructions.

```
//Arithmetic pseudo-instructions

class UVE_PSEUDO_V_VVVP : Pseudo<(outs UR:$ud), (ins UR:$us1, UR:$us2, UR
    :$us3, UPR_3b:$ps1), [], "", "">;

multiclass UVE_PSEUDO_ARITH_V_VVVP_USF{
  def _U        : UVE_PSEUDO_V_VVVP;
  def _S        : UVE_PSEUDO_V_VVVP;
  def _FP       : UVE_PSEUDO_V_VVVP;
}
let hasSideEffects = 0, mayLoad = 1, mayStore = 1 in {

  defm UVE_PSEUDO_ADD   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_SUB   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_MUL   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_DIV   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_MAC   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_ABS   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_MIN   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  defm UVE_PSEUDO_MAX   :  UVE_PSEUDO_ARITH_V_VVVP_USF;
  ...
}
```

Like it was done with the instructions, a pattern is created to link between the intrinsic and corresponding pseudo-instruction, so that during instruction selection the pseudo-instruction replaces the intrinsic. The code to generate this patterns follow the same behaviour as shown before, pattern with the intrinsic as the first operand of *Pat* and corresponding pseudo-instruction as the second operand.

The code generation process will go through all the stages, including register allocation, and then it will find the pass *ExpandPostRAPseudos.cpp*. This pass will delegate the job of expanding the pseudo-instruction to the targets, if it find any. In `llvm/lib/Target/RISCV/RISCVInstrInfo.cpp`, the method to expand the pseudo-instruction should be implemented as shown in Listing 3.20. The function *expandPostRAPseudo* will be called from outside and will check the Machine Instruction Opcodes againt the pseudo-instruction Opcodes. If it finds any pseudo-instruction, it expands it by creating a new Machine Instruction in the place of the pseudo-instruction with the additional operator that was added before (the stream register) as the output of the Machine Instruction.

Listing 3.20: Expanding pseudo-instruction that matches with *UVE_ABS _FP instruction*.

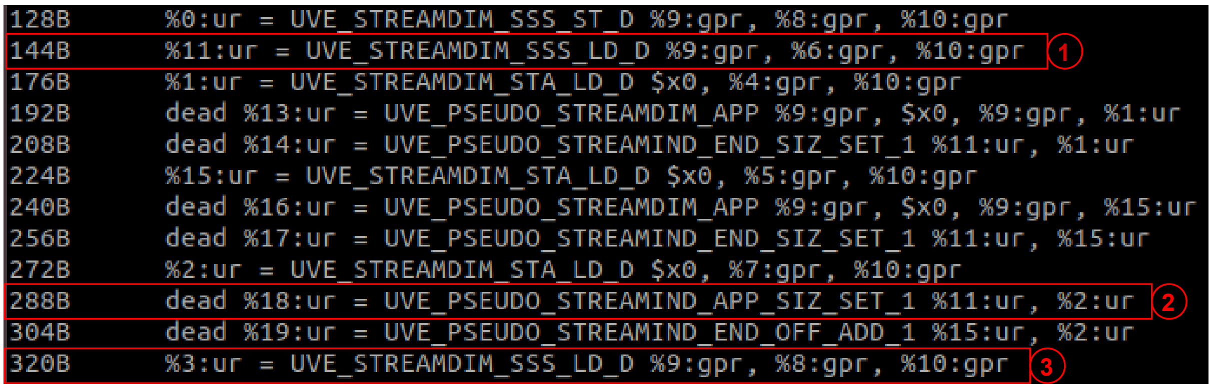
```
static bool expandPseudo_V_VVVP(MachineInstr &MI, const MCInstrDesc &Desc) {
  MachineInstrBuilder MIB(*MI.getParent()->getParent(), MI);

  MachineBasicBlock &MBB = *MIB->getParent();
  DebugLoc DL = MIB->getDebugLoc();
  Register Dest = MIB.getReg(3);

  BuildMI(MBB, MIB.getInstr(), DL, Desc, Dest).addReg(MIB.getReg(1))
                          .addReg(MIB.getReg(2)).addReg(MIB.getReg(4));

  MBB.erase(MI);

  return true;
}

bool RISCVInstrInfo::expandPostRAPseudo(MachineInstr &MI) const {

  switch (MI.getOpcode()) {
  case RISCV::UVE_PSEUDO_ABS_FP :
    return expandPseudo_V_VVVP(MI, get(RISCV::UVE_ABS_FP));
  ...
}
```

3.2.5 Register Coalescing

Even though all the streams are defined with UVE instructions, they are defined implicitly. That is, to define a stream, a instruction receives some parameters to configure that stream and then writes the output register. This output register is implicitly the stream. However, this presents a problem: if a register configured as a stream is only used, as an example, to configure another stream by an indirect modifier and never used inside an instruction it might be rewritten by the compiler. The reason for that is when the compiler is doing register allocation it tries to minimize the amount of registers used. To do that, it first assigns live ranges to all the registers, one by one. Live ranges are used to know where in the program a register is in use and when it no longer will be used throughout the program. Then, if a register is no longer in use, the register allocator might perform register coalescing, that is, it allocates two variable for the same register.

Figure 3.2 and Figure 3.3 illustrate the problem described previously.

Figure 3.2 presents a representation of the instructions during register allocation. At this stage,



```
128B    %0:ur = UVE_STREAMDIM_SSS_ST_D %9:gpr, %8:gpr, %10:gpr
144B    %11:ur = UVE_STREAMDIM_SSS_LD_D %9:gpr, %6:gpr, %10:gpr  1
176B    %1:ur = UVE_STREAMDIM_STA_LD_D $x0, %4:gpr, %10:gpr
192B    dead %13:ur = UVE_PSEUDO_STREAMDIM_APP %9:gpr, $x0, %9:gpr, %1:ur
208B    dead %14:ur = UVE_PSEUDO_STREAMIND_END_SIZ_SET_1 %11:ur, %1:ur
224B    %15:ur = UVE_STREAMDIM_STA_LD_D $x0, %5:gpr, %10:gpr
240B    dead %16:ur = UVE_PSEUDO_STREAMDIM_APP %9:gpr, $x0, %9:gpr, %15:ur
256B    dead %17:ur = UVE_PSEUDO_STREAMIND_END_SIZ_SET_1 %11:ur, %15:ur
272B    %2:ur = UVE_STREAMDIM_STA_LD_D $x0, %7:gpr, %10:gpr
288B    dead %18:ur = UVE_PSEUDO_STREAMIND_APP_SIZ_SET_1 %11:ur, %2:ur  2
304B    dead %19:ur = UVE_PSEUDO_STREAMIND_END_OFF_ADD_1 %15:ur, %2:ur
320B    %3:ur = UVE_STREAMDIM_SSS_LD_D %9:gpr, %8:gpr, %10:gpr  3
```

Figure 3.2: Example Machine Instructions and corresponding input and output variables during register allocation.

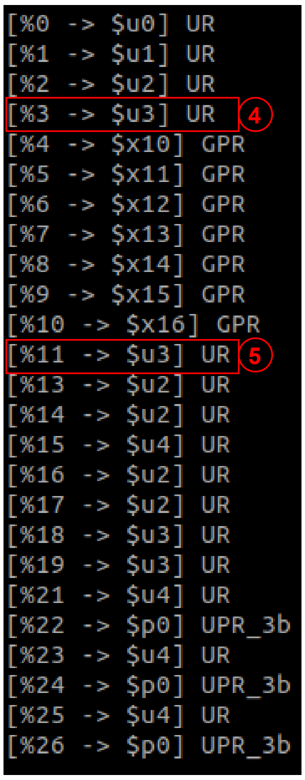
they are represented as Machine Instructions and each input and output parameter is associated with a variable, represented by the string ”%” followed by a number. Highlighted with the number 1 is an UVE instruction defining a stream, that uses as an output variable ”%11”. This stream is later used to define an indirect modifier, highlighted with the number 2. Number 3 is another definition of a separate stream, that has no connection to the one defined in 1.

Figure 3.3 illustrates the mapping between the variables used as parameters for the Machine Instructions and the register used. Because the stream identified in 1 will no longer be used throughout the rest of the program, this register will be rewritten by another stream, in this case the one defined in 3. Highlighted in number 4 is the mapping between variable ”%3” and UVE ”u3” scalable register and in number 5 is the mapping between a different variable ”%11” and the same UVE scalable register ”u3”.

The first solution that seemed more obvious was to change the pass that implements the register allocator or rewrite one from scratch to detect when a variable is associated with a stream and stop is from rewriting it. Both of this solutions are really complex and time consuming, so another method would be much more appreciated.

The second solution, and the one implemented in this work, is again the use of pseudo-instructions. The idea is to define a new pseudo-instruction and a new intrinsic to match it that only serves as a stopper for register coalescing. The intrinsic is called when a stream is no longer necessary, and the corresponding register can be rewritten. This stops the register allocator from coalescing this register because the variable associated with the stream will still be in use up until this pseudo-instruction. Because pseudo-instructions are only expanded after register allocation, it is then safe to just remove the instruction and the register will no longer be rewritten by another stream.

The new intrinsic is defined as *llvm.riscv.uve.freeze* in LLVM IR and takes a single input, the scalable register that contain the stream the user doesn’t want to be rewritten up until that point.



```
[%0 -> $u0] UR
[%1 -> $u1] UR
[%2 -> $u2] UR
[%3 -> $u3] UR
[%4 -> $x10] GPR
[%5 -> $x11] GPR
[%6 -> $x12] GPR
[%7 -> $x13] GPR
[%8 -> $x14] GPR
[%9 -> $x15] GPR
[%10 -> $x16] GPR
[%11 -> $u3] UR
[%13 -> $u2] UR
[%14 -> $u2] UR
[%15 -> $u4] UR
[%16 -> $u2] UR
[%17 -> $u2] UR
[%18 -> $u3] UR
[%19 -> $u3] UR
[%21 -> $u4] UR
[%22 -> $p0] UPR_3b
[%23 -> $u4] UR
[%24 -> $p0] UPR_3b
[%25 -> $u4] UR
[%26 -> $p0] UPR_3b
```

Figure 3.3: Example register mapping of the variables defined in Figure 3.2.

3.3 Summary

At the start of this chapter a list of requirements for the supporting LLVM extension was defined, to guide its implementation. To represent scalable vectors on the backend and IR, it was used an already defined representation that was developed by Arm for its SVE extension. This type defines the minimum number of elements in the scalable vector and the type of its elements, while also signaling it is scalable. It was also created three new register classes, that encompass the scalable registers, the full 16 scalable predicate registers and one with only the 8 scalable predicate registers, from 0 to 7. The definition of UVE instructions was done by instantiating the *RVInst* class and encoding the right parameters according to the instruction format. To overcome the limitations presented by the LLVM IR SSA form, pseudo-instructions were defined as placeholders, to replace them late in the code generation pipeline by the correct instruction ignoring the SSA limitations and using one of the inputs to the pseudo-instruction as the return type, that way being able to write to already configured streams. To represent UVE instructions in LLVM IR intrinsics were defined. They act as a C function, taking inputs and outputs, and are transparent to optimization passes. To link intrinsics with instructions and pseudo-instructions patterns were used. Each pattern defines, as the name suggests, a pattern that may occur in the representation to later be replaced during instruction selection by one or more instructions. In this case, most of the patterns were defined as a one to one, from one intrinsic to one instruction/pseudo-instruction. Finally, register coalescing was solved by implementing a simple pseudo-instruction that serves as a placeholder to stop the register allocator from rewriting registers associated with streams.

This implementation, with all the characteristics stated above, fulfills the requirements that were defined during Section 3.1.

4

Results and Discussion

Contents

4.1 Evaluation Method

To evaluate if the implementations is working as intended and is able to translate LLVM IR into target UVE code, LLVM's *llc* tool will be used. *Llc* is LLVM's static compiler, and because a linker for this extension exists separately and is not yet implemented into LLVM, the goal of this evaluation is to check if the llc tool can compile from LLVM IR using the previously defined intrinsics into an assembly representation that uses UVE's instructions encoding and nomenclature.

To perform such evaluation, three benchmarks will be used:

- **Single-Precision A· X Plus Y (SAXPY) Kernel** : This is a simple benchmark that performs the operations stated in its name. This benchmark is used as an introduction, to test the definition of single dimension streams and arithmetic instructions.

- **Trisolv Kernel** : Trisolv is a triangular solver that is part of the PolyBench collection of benchmarks. It introduces multi dimensional data accesses and triangular matrix accesses and will be used to test the multi dimensional implementation, including the overcome of the SSA form, and also the use of simple modifiers, to describe the triangular matrix pattern.

- **Sparse Matrix-Vector Multiply (SPMV) Kernel** : SPMV is a common operation among scientific codes and is used in iterative methods to solve sparse linear systems. This kernel introduces indirect memory accesses, that will be used to test the implementation of indirect modifiers. It will also present a situation where the freezing intrinsics are necessary.

4.2 Benchmarks Evaluation

The benchmarks evaluation will be done in the order they were introduced in the previous section. First, it will be introduced the C code representation, followed by the IR counterpart and the result produced by the static compiler *llc*.

The IR representation that will be shown is an hand written version, as there is no frontend that supports the custom LLVM instructions and was also not implemented in this work.

4.2.1 SAXPY

As described before, SAXPY kernel presents a simple structure, described in Listing 4.1.

Listing 4.1: SAXPY kernel described in C language.

```c
kernel_saxpy(DataType *x, DataType *y, DataType A, int sizeN) {

    for (int i = 0; i < sizeN; i++) {
        y[i] += x[i] * A;
    }
    return;
}
```

Variables *y* and *x* can be described as streams with only one dimension, while the variable A will need to be duplicated into a scalable vectorial register to perform the arithmetic operations. LLVM IR representation is listed in Listing 4.2.

Listing 4.2: SAXPY kernel described in LLVM IR.

```
1  ...
2  define dso_local void @kernel_saxpy(i64* %x, i64* %y, i64 %A, i64 %sizeN)
       #0 {
3  entry:
4    %streamx = call <vscale x 1 x i64> @llvm.riscv.uve.stream.dim.sta.ld.d(
       i64 %sizeN, i64* %x, i64 1)
5    %streamyL = call <vscale x 1 x i64> @llvm.riscv.uve.stream.dim.sta.ld.d
       (i64 %sizeN, i64* %y, i64 1)
6    %streamyS = call <vscale x 1 x i64> @llvm.riscv.uve.stream.dim.sta.st.d
       (i64 %sizeN, i64* %y, i64 1)
7    %streamA = call <vscale x 1 x i64> @llvm.riscv.uve.move.duplicate.d(i64
       %A, <vscale x 2 x i1> undef)
8    br label %loop
9  loop:
10   %tempStream = call <vscale x 1 x i64> @llvm.riscv.uve.mul.s.nxv1i64(<
       vscale x 1 x i64> %streamx, <vscale x 1 x i64> %streamA, <vscale x 2
       x i1> undef)
11   %dummy1 = call <vscale x 1 x i64> @llvm.riscv.uve.add.s.save.nxv1i64(<
       vscale x 1 x i64> %streamyL, <vscale x 1 x i64> %tempStream, <vscale
       x 1 x i64> %streamyS, <vscale x 2 x i1> undef)
12   %loopRes = call i64 @llvm.riscv.uve.branch.comp.1.nxv1i64(<vscale x 1 x
       i64> %streamx)
13   %branch1 = trunc i64 %loopRes to i1
14   br i1 %branch1, label %loop, label %return_label
15 return_label:
16   ret void
17 }
18 ...
```

On line 1 the three dots are in place of file identification, such as the source file it derives from and the target layout and triple, that were already explained before. On line 18 the three dots are in place of the intrinsics declarations and metadata definition. All the elements removed add nothing to the evaluation.

The definition of streams is done on lines 4, 5 and 6. Is is necessary both a load stream and a store stream to represent the memory accesses done by the *y* variable because of the sum connected to the equality. On line 11 the code calls for an UVE arithmetic add that saves the result into the third parameter, the variable representing the *y* storing stream. In this example there is no need to make use of the freeze instruction as the registers will never be rewritten. Predicates are left undefined using the *undef* keyword, as they fallback to the scalable predicate register *po*, enabling all lanes.

The result from the compiler processing is displayed in Listing 4.3. The corresponding encoding to each instruction is displayed by its side.

There is no register rewriting, as expected, and the arithmetic instruction on line 18 stores the result on register *u2*, by making use of pseudo-instructions. The *fixup* that was defined particularly for UVE branches is also displayed on line 20. Overall, the instructions were successfully compiled into assembly and the defined streams were unchanged throughout all the program's duration.

Listing 4.3: SAXPY kernel assembly format after compilation.

```
 1 ...
 2 kernel_saxpy:                                  # @kernel_saxpy
 3 # %bb.0:                                       # %entry
 4   addi  sp, sp, -16                            # encoding: [0x41,0x11]
 5   sd  ra, 8(sp)                              # encoding: [0x06,0xe4]
 6   sd  s0, 0(sp)                              # encoding: [0x22,0xe0]
 7   addi  s0, sp, 16                             # encoding: [0x00,0x08]
 8   addi  a4, zero, 1                            # encoding: [0x05,0x47]
 9   ss.sta.ld.d u0, a3, a0, a4                 # encoding: [0x0b,0xf0,0xa6,0x74]
10   ss.sta.ld.d u1, a3, a1, a4                 # encoding: [0x8b,0xf0,0xb6,0x74]
11   ss.sta.st.d u2, a3, a1, a4                 # encoding: [0x0b,0xb1,0xb6,0x74]
12   so.v.dp.d u3, a2, p0                     # encoding: [0xab,0x31,0x06,0xac]
13   j .LBB0_1                                # encoding: [0bAAAAAA01,0b101AAAAA]
14                                                #   fixup A - offset: 0, value:
                                  .LBB0_1, kind: fixup_riscv_rvc_jump
15 .LBB0_1:                                       # %loop
16                                                # =>This Inner Loop Header: Depth
     =1
17   so.a.mul.sg u4, u0, u3, p0                 # encoding: [0x2b,0x22,0x30,0x10]
18   so.a.add.sg u2, u1, u4, p0                 # encoding: [0x2b,0xa1,0x40,0x00]
19   so.b.dc.1 u0, .LBB0_1                    # encoding: [0x2b'A',A,A,0xe0'A']
20                                                #   fixup A - offset: 0, value:
                      .LBB0_1, kind: fixup_riscv_uve_stream_branch
21   j .LBB0_2                                # encoding: [0bAAAAAA01,0b101AAAAA]
22                                                #   fixup A - offset: 0, value:
                                  .LBB0_2, kind: fixup_riscv_rvc_jump
23 .LBB0_2:                                       # %return_label
24   ld  s0, 0(sp)                              # encoding: [0x02,0x64]
25   ld  ra, 8(sp)                              # encoding: [0xa2,0x60]
26   addi  sp, sp, 16                             # encoding: [0x41,0x01]
27   ret                                            # encoding: [0x82,0x80]
28 .Lfunc_end0:
29   .size kernel_saxpy, .Lfunc_end0-kernel_saxpy
30 ...
```

4.2.2 Trisolv

Trisolv kernel is displayed in Listing 4.4, presented in the C language.

Listing 4.4: Trisolv kernel described in C language.

```
kernel_trisolv(DataType **L, DataType *b, DataType *x, int sizeN) {

  int i,j;

  for (i = 0; i < sizeN; i++)
    {
      x[i] = b[i];
      for (j = 0; j < i; j++)
        x[i] -= L[i][j] * x[j];

      x[i] = x[i] / L[i][i];
    }
}
```

This kernel presents some added complexity in comparison to the previous one. It features multi dimensional memory accesses and a triangular matrix pattern. To describe such pattern it is necessary to make use of direct modifiers. The LLVM IR representation of the same kernel is presented in Listing 4.5. Only a part of the code is shown, as it is slightly extensive and most of it presents no new features in comparison to SAXPY. Only new features and instructions that aid in comprehension are presented.

Listing 4.5: Trisolv kernel described in LLVM IR.

```
 1 ...
 2 define dso_local void @kernel_trisolv(i64 **%L, i64 *%b, i64 *%x, i64 %
       sizeN) #0 {
 3 entry:
 4   %streamxiStore = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sss.st.d(i64 %sizeN, i64* %x, i64 1)
 5   %streambiLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sss.ld.d(i64 %sizeN, i64* %b, i64 1)
 6   %streamxjLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sta.ld.d.1p(i64 0, i64* %x, i64 1)
 7   %dummy2 = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.mod.end.siz.inc.nxv1i64(i64 %sizeN, i64 1, <
       vscale x 1 x i64> %streamxjLoad)
 8   %streamliiLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sta.ld.d.2p(i64 %sizeN, i64** %L, i64 1)
 9   %dummy3 = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.end.nxv1i64(i64 %sizeN, i64* null, i64 %
       sizeN, <vscale x 1 x i64> %streamliiLoad)
10   %streamlijLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sta.ld.d.2p(i64 0, i64** %L, i64 1)
11   %dummy4 = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.app.nxv1i64(i64 %sizeN, i64* null, i64 %
       sizeN, <vscale x 1 x i64> %streamlijLoad)
12   %dummy5 = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.mod.end.siz.inc.nxv1i64(i64 %sizeN, i64 1, <
       vscale x 1 x i64> %streamlijLoad)
13   %streamxiLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sss.ld.d(i64 %sizeN, i64* %x, i64 1)
14   br label %loop1
15 loop1:
16 ...
```

Arithmetic intrinsics always follow the same format, so there was no need to include them in Listing 4.5. The main features of this kernel can be presented during stream configuration. On line 7 is the definition of the stream's first dimension, corresponding to C's memory access L[i][i]. The second dimension is appended on line 7, and receives as argument the returning value of line 6, to later use the same scalable register for all stream definitions. The description of the memory access L[i][j] is done on lines 10, 11 and 12. Two dimensions are defined as before but the first one starts with a size of 0. This size will be incremented every time dimension 2 is iterated by the modifier defined on line 12.

The compiled assembly representation is in Listing 4.6. It is also cropped, with only the code produced from Listing 4.5 compilation.

Again, all the intrinsics were translated successfully into the target's assembly. The double dimension definitions can be seen on lines 12 and 13 for the L[i][i] memory access and lines 14 and 15 for the L[i][j] memory access. The direct modifier for this last stream is defined on line 16. There, the size of the first dimension is increased incrementally one step each time the second dimension is iterated.

4.2.3 SPMV

The last kernel to be evaluated. To better explain the intricacies of such patterns, the C language representation is presented in Listing 4.7. In this kernel, the inner loop is controlled by an iteration of the variable *nnz_A*. This implies that every other memory access that depends on the inner loop iterations

Listing 4.6: Trisolv kernel assembly format after compilation.

```
...
kernel_trisolv:                                      # @kernel_trisolv
# %bb.0:                                             # %entry
    addi   sp, sp, -16                               # encoding: [0x41,0x11]
    sd   ra, 8(sp)                                   # encoding: [0x06,0xe4]
    sd   s0, 0(sp)                                   # encoding: [0x22,0xe0]
    addi   s0, sp, 16                                # encoding: [0x00,0x08]
    addi   a4, zero, 1                               # encoding: [0x05,0x47]
    ss.st.d u0, a3, a2, a4                           # encoding: [0x0b,0xb0,0xc6,0x76]
    ss.ld.d u1, a3, a1, a4                           # encoding: [0x8b,0xf0,0xb6,0x76]
    ss.sta.ld.d u2, zero, a2, a4                     # encoding: [0x0b,0x71,0xc0,0x74]
    ss.end.mod.siz.inc  u2, a3, a4                   # encoding: [0x0b,0xc1,0x06,0x72]
    ss.sta.ld.d u3, a3, a0, a4                       # encoding: [0x8b,0xf1,0xa6,0x74]
    ss.end  u3, a3, zero, a3                         # encoding: [0x8b,0x81,0x06,0x6a]
    ss.sta.ld.d u4, zero, a0, a4                     # encoding: [0x0b,0x72,0xa0,0x74]
    ss.app  u4, a3, zero, a3                         # encoding: [0x0b,0x82,0x06,0x68]
    ss.end.mod.siz.inc  u4, a3, a4                   # encoding: [0x0b,0xc2,0x06,0x72]
    ss.ld.d u5, a3, a2, a4                           # encoding: [0x8b,0xf2,0xc6,0x76]
    j .LBB0_1                                      # encoding: [0bAAAAAA01,0b101AAAAA]
                                                       #   fixup A - offset: 0, value:
                                               .LBB0_1, kind: fixup_riscv_rvc_jump
.LBB0_1:                                               # %loop1
...
```

Listing 4.7: SPMV kernel described in C language.

```c
kernel_spmv(DataType **vals_A, int **idx_A, int *nnz_A,
            DataType *x, DataType *y, int sizeN) {

  DataType sum = 0;

  for (int i=0; i < sizeN; i++)  {
    y[i] = 0;
    for (int j = 0; j < nnz_A[i]; j++)
      y[i] +=  vals_A[i][j] * x[ idx_A[i][j] ];
  }
}
```

will also be dependent on the control variable itself, thus creating and indirect dependency. The variable *x* also presents an interesting case where both the number of iterations and the offset from the base memory will be dependent on two distinct variables, *nnz_A* and *idx_A* respectively. The corresponding LLVM IR representation is in Listing 4.8.

The main features presented in Listing 4.8 are the use of indirect modifiers and the presence of the freeze intrinsics. The indirect modifier is featured on line 8. This indirect modifier is applied every time the second dimension on line 7 is iterated and it sets the size of the first dimension to the value of stream defined on line 5. The double dependency described before on variable *x* can be seen also defined by indirect modifiers on lines 12 and 13. This variable is dependent on two different variable, however, as it is index by the variable *idx_A* and this variable is also dependent on the *nnz_A* variable to determine its size, the size of *x* will be, in reality, defined by the indirect modifier on line 11. The second feature described in this kernel is the presence of freeze intrinsics. There is one for each defined stream at the end of the kernel, only ”releasing” the scalable vector at the end. This intrinsics serve only as placeholders and will not be present on the compiled output. Listing 4.9 shows the compiled file.

Listing 4.8: SPMV kernel described in LLVM IR.

```
...
define dso_local void @kernel_smpv(i64 **%vals_A, i64 **%idx_A, i64 *%
    nnz_A, i64 *%x, i64 *%y, i64 %sizeN) #0 {
entry:
  %streamyStore = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.sss.st.d(i64 %sizeN, i64* %y, i64 1)
  %streamnnz_ALoad = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.sss.ld.d(i64 %sizeN, i64* %nnz_A, i64 1)
  %streamvals_ALoad = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.sta.ld.d.2p(i64 0, i64** %vals_A, i64 1)
  %dummy1 = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.app.nxv1i64(i64 %sizeN, i64* null, i64 %
    sizeN, <vscale x 1 x i64> %streamvals_ALoad)
  %dummy2 = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.ind.end.siz.set.1.nxv1i64(<vscale x 1 x i64> %
    streamnnz_ALoad, <vscale x 1 x i64> %streamvals_ALoad)
  %streamidx_ALoad = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.sta.ld.d.2p(i64 0, i64** %idx_A, i64 1)
  %dummy3 = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.app.nxv1i64(i64 %sizeN, i64* null, i64 %
    sizeN, <vscale x 1 x i64> %streamidx_ALoad)
  %dummy4 = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.ind.end.siz.set.1.nxv1i64(<vscale x 1 x i64> %
    streamnnz_ALoad, <vscale x 1 x i64> %streamidx_ALoad)
  %streamxLoad = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.sta.ld.d.1p(i64 0, i64* %x, i64 1)
  %dummy6 = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.ind.end.off.add.1.nxv1i64(<vscale x 1 x i64> %
    streamidx_ALoad, <vscale x 1 x i64> %streamxLoad)
  %streamyLoad = call <vscale x 1 x i64>
    @llvm.riscv.uve.stream.dim.sss.ld.d(i64 %sizeN, i64* %y, i64 1)
  br label %loop1
loop1:
  ...
return_label:
  call void @llvm.riscv.uve.freeze(<vscale x 1 x i64> %streamyStore)
  call void @llvm.riscv.uve.freeze(<vscale x 1 x i64> %streamnnz_ALoad)
  call void @llvm.riscv.uve.freeze(<vscale x 1 x i64> %streamvals_ALoad)
  call void @llvm.riscv.uve.freeze(<vscale x 1 x i64> %streamidx_ALoad)
  call void @llvm.riscv.uve.freeze(<vscale x 1 x i64> %streamxLoad)
  call void @llvm.riscv.uve.freeze(<vscale x 1 x i64> %streamyLoad)
  ret void
}
...
```

Listing 4.9: SPMV kernel assembly format after compilation.

```
1  ...
2  kernel_smpv:                              # @kernel_smpv
3  # %bb.0:                                  # %entry
4    addi   sp, sp, -16                      # encoding: [0x41,0x11]
5    sd   ra, 8(sp)                        # encoding: [0x06,0xe4]
6    sd   s0, 0(sp)                        # encoding: [0x22,0xe0]
7    addi   s0, sp, 16                       # encoding: [0x00,0x08]
8    addi   a6, zero, 1                      # encoding: [0x05,0x48]
9    ss.st.d u0, a5, a4, a6                    # encoding: [0x0b,0xb0,0xe7,0
     x86]
10   ss.ld.d u1, a5, a2, a6                    # encoding: [0x8b,0xf0,0xc7,0
     x86]
11   ss.sta.ld.d u2, zero, a0, a6          # encoding: [0x0b,0x71,0xa0,0x84]
12   ss.app  u2, a5, zero, a5                  # encoding: [0x0b,0x81,0x07,0
     x78]
13   ss.end.ind.siz.set.1  u2, u1            # encoding: [0x0b,0xe1,0x00,0x03]
14   ss.sta.ld.d u3, zero, a1, a6          # encoding: [0x8b,0x71,0xb0,0x84]
15   ss.app  u3, a5, zero, a5                  # encoding: [0x8b,0x81,0x07,0
     x78]
16   ss.end.ind.siz.set.1  u3, u1            # encoding: [0x8b,0xe1,0x00,0x03]
17   ss.sta.ld.d u4, zero, a3, a6          # encoding: [0x0b,0x72,0xd0,0x84]
18   ss.end.ind.off.add.1  u4, u3            # encoding: [0x0b,0xe2,0x21,0x02]
19   ss.ld.d u5, a5, a4, a6                    # encoding: [0x8b,0xf2,0xe7,0
     x86]
20   j .LBB0_1                           # encoding: [0bAAAAAA01,0b101AAAAA]
21                                           #   fixup A - offset: 0, value:
     .LBB0_1, kind: fixup_riscv_rvc_jump
22 .LBB0_1:                                  # %loop1
23 ...
24 .LBB0_4:                                  # %return_label
25   ld   s0, 0(sp)                        # encoding: [0x02,0x64]
26   ld   ra, 8(sp)                        # encoding: [0xa2,0x60]
27   addi   sp, sp, 16                       # encoding: [0x41,0x01]
28   ret                                       # encoding: [0x82,0x80]
29 .Lfunc_end0:
```

As it can be seen in Listing 4.9, there is no register rewriting on top of any register defined as a stream. The only rewriting that happens is to configure the dimensions and indirect modifiers, as expected. Another thing to point out is the absence of the freezing pseudo-instructions on the last loop on line 24, now labeled *.LBB0_4*, which means the freezing intrinsics are working as intended.

To consolidate the necessity of the freezing intrinsics at the end to avoid register coalescing, the same kernel was compiled but now without the freezing intrinsics, and the result is in listing 4.10. As it can be seen on lines 17 and 19, both ”u2” and ”u3” scalable registers are being written on top of the streams defined on lines 10 and 14, respectively. Although this streams are no longer used anywhere else on the program, they are still used implicitly by the streaming unit to gather data for the streams configured with indirect modifiers. That is why it is necessary to use freezing intrinsics at the end in listing 4.8.

Overall, the kernels performed as what was excepted during implementation and fulfilled the requirements listed in Section 3.1.

Listing 4.10: SPMV kernel assembly format without freezing intrinsics after compilation.

```
...
kernel_smpv:                                # @kernel_smpv
# %bb.0:                                    # %entry
  addi  sp, sp, -16                         # encoding: [0x41,0x11]
  sd  ra, 8(sp)                           # encoding: [0x06,0xe4]
  sd  s0, 0(sp)                           # encoding: [0x22,0xe0]
  addi  s0, sp, 16                          # encoding: [0x00,0x08]
  addi  a6, zero, 1                         # encoding: [0x05,0x48]
  ss.st.d u0, a5, a4, a6                      # encoding: [0x0b,0xb0,0xe7,0
    x86]
  ss.ld.d u2, a5, a2, a6                      # encoding: [0x0b,0xf1,0xc7,0
    x86]
  ss.sta.ld.d u1, zero, a0, a6            # encoding: [0x8b,0x70,0xa0,0x84]
  ss.app  u1, a5, zero, a5                    # encoding: [0x8b,0x80,0x07,0
    x78]
  ss.end.ind.siz.set.1  u1, u2              # encoding: [0x8b,0x60,0x01,0x03]
  ss.sta.ld.d u3, zero, a1, a6            # encoding: [0x8b,0x71,0xb0,0x84]
  ss.app  u3, a5, zero, a5                    # encoding: [0x8b,0x81,0x07,0
    x78]
  ss.end.ind.siz.set.1  u3, u2              # encoding: [0x8b,0x61,0x01,0x03]
  ss.sta.ld.d u2, zero, a3, a6            # encoding: [0x0b,0x71,0xd0,0x84]
  ss.end.ind.off.add.1  u2, u3              # encoding: [0x0b,0xe1,0x21,0x02]
  ss.ld.d u3, a5, a4, a6                      # encoding: [0x8b,0xf1,0xe7,0
    x86]
  j .LBB0_1                             # encoding: [0bAAAAAA01,0b101AAAAA]
                                            #   fixup A - offset: 0, value:
    .LBB0_1, kind: fixup_riscv_rvc_jump
.LBB0_1:                                    # %loop1
...
.LBB0_4:                                    # %return_label
  ld  s0, 0(sp)                           # encoding: [0x02,0x64]
  ld  ra, 8(sp)                           # encoding: [0xa2,0x60]
  addi  sp, sp, 16                          # encoding: [0x41,0x01]
  ret                                         # encoding: [0x82,0x80]
.Lfunc_end0:
```

4.3 Summary

At the start of this section a methodology was defined to evaluate the extension. Such evaluation is based on compiling three kernels with different features that put to test all that was implemented for this compiler extension. The analysis was done by evaluating how easy and similar it is to use the implemented intrinsics, although LLVM IR uses a complete different format (SSA), and how the compiled code looked like what was expected to be. The first kernel used for testing was SAXPY, that presents a simple structure and puts to test the ability to define streams and perform arithmetic and vector manipulation operations. Following SAXPY kernel came trisolv kernel, that presents a more complex structure with multi dimensional streams and iterations controlled by an incremental variable, forcing the use of direct modifiers. The last kernel used was SPMV, and this one presented an opportunity to use the freezing intrinsic to avoid register coalescing during register allocation and also indirect modifiers. All the benchmarks were able to be represented in LLVM IR using the implemented features and produced the expected assembly code.

5 Conclusions and Future Work

Contents

5.1 Conclusions

The new experimental scalable extension UVE to RISC V architecture presents an exciting alternative that combines the advantages of scalable vectorial architectures with the streaming paradigm. This results in the emission of less instructions, that lead to less clock cycles to process all the instructions. This extension also features memory decoupling from the main processing unit, leaving all the memory operation related to streams for a separate streaming unit. By describing the memory access patterns done by the variables inside loops, the streaming unit takes care of prefetching the necessary data to be ready for processing when the instructions are issued.

To support the UVE extension, it is implemented an extension for the LLVM Infrastructure. As the LLVM backend presents a modular structure, the implementation can be contained to the architecture it targets, RISC V. The backend is populated with new instruction encoded with their respective formats and intrinsics functions are used to represent such instructions in LLVM IR. Although the SSA format is incompatible with format used by UVE instructions, pseudo-instructions are used as placeholders, to be replaced by the correct instructions and formats after register allocation. To avoid unwanted register coalescing by the compiler during register allocation, a freezing intrinsic is used to lock a register up until that point in the code, later being removed entirely.

The supporting implementation is able to represent the new instruction in LLVM IR format through intrinsics and compiles the three tested kernels that are representative of a big part of the implementation features. They are all able to be compiled without any errors and produce assembly code that is according to the UVE standard.

5.2 Future Work

As a functional backend for the UVE extension, this work still leaves room for some opportunities of future work.

While this work implements a backend that is able to compile into the target assembly, it does not offer any frontend support. Such could be a topic of a future work. By taking advantage of the Clang frontend, it could implement the logic necessary to translate from C source code into LLVM IR.

On the other end, the LLVM implementation could also be supplemented with the integration of a specific target assembler. This way, in conjunction with the the frontend, it would be able to directly compile form C/C++ source code into the native object file, only needing a linker at the end and it would make the compilation process faster, by avoiding having to parse and encode the assembly instructions again.

Another interesting opportunity that is left open is the investigation and development of techniques for optimization of the UVE code generation process. However, LLVM intrinsics are transparent to optimization passes, so the targets of such optimizations would likely have to be implemented as LLVM IR instructions instead of intrinsics, which is a problem of its own. By analysing the intermediate representations, not on the LLVM IR but also others that are used throughout the compiler pipeline, such as the DAG used during code generation and the AST on the frontend, it could also be implemented some kind of auto vectorization that detects opportunities to transform normal loop instructions into UVE ones.

Bibliography

[1] N. Neves, "Energy-Efficient Computing: Adaptive Structures and Data Management," Ph.D. dissertation, Instituto Superior Técnico, Universidade de Lisboa, 1 2019.

[2] J. Domingos, "Unlimited Vector Extension with data streaming support," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2020.

[3] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual.* CS Division, EECS Department, University of California, Berkeley, 5 2017, vol. I, no. 2.2.

[4] R. Ramanathan, R. Curry, S. Chennupaty, R. L. Cross, S. Kuo, and M. J. Buxton, "Extending the World's Most Popular Processor Architecture," White Paper, Tech. Rep., 2006. [Online]. Available: http://www.ele.uva.es/~jesman/BigSeti/ftp/Microprocesadores/Intel/new-instructions-paper.pdf

[5] C. Lomont, "Introduction to Intel Advanced Vector Extensions," White Paper, Tech. Rep., 2011. [Online]. Available: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf

[6] ARM, "Introducing NEON™ Development Article," Tech. Rep., 2009. [Online]. Available: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON?lang=en

[7] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[8] M. V. Wilkes, "Computers Then and Now," *Journal of the Association for Computing Machinery*, pp. 1–7, 1 1968.

[9] B. et al., "The FORTRAN automatic coding system," Western Joint Computer Conference. Spartan Books, 1957, p. 188–198.

[10] C. Severance and K. Dowd, *What a Compiler Does - History of Compilers*, pp. 47–48. [Online]. Available: http://cnx.org/content/m33686/1.3/

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed. Addison Wesley, 8 2006.

[12] K. Schwarz, “Lexical Analysis.” [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/01/Slides01.pdf

[13] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and V. Huy, *The Architecture of Open Source Applications*, 01 2011, pp. 155–171.

[14] A. Amid, K. Asanovic, A. Baum, A. Bradbury, T. Brewer, C. Celio, A. Chapyzhenka, S. Chiricescu, K. Dockser, and B. Dreyer, “RISC-V ”V” Vector Extension,” Tech. Rep., 2019. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

[15] G. Hunter and A. Emerson, “Scalable Vectorization in LLVM,” ARM, Tech. Rep., 11 2016. [Online]. Available: https://llvm.org/devmtg/2016-11/Slides/Emerson-ScalableVectorizationinLLVMIR.pdf

[16] B. S. Center and SiFive, “Code generation for RISC-V V-extension,” 10 2020. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2020-October/145850.html

[17] SiFive and B. S. Center, “RISC-V Vector Extension Intrinsic Document,” 2021. [Online]. Available: https://github.com/riscv-non-isa/rvv-intrinsic-doc

[18] N. Neves, P. Tomás, and N. Roma, “Adaptive In-Cache Streaming for Efficient Data Management,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 7, pp. 2130–2143, 2017.

[19] N. Neves, N. Roma, and P. Tomás, “Efficient data-stream management for shared-memory many-core systems,” in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[20] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner, “Imagine: media processing with streams,” *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[21] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, “The reconfigurable streaming vector processor (rsvp/spl trade/),” in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 141–150.

[22] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, “Q100: The architecture and design of a database processing unit,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 255–268. [Online]. Available: https://doi.org/10.1145/2541940.2541961

[23] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 416–429.

[24] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *2008 International Symposium on Computer Architecture*, 2008, pp. 389–400.

[25] G. Weisz and J. C. Hoe, "Coram++: Supporting data-structure-specific memory interfaces for fpga computing," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[26] C.-H. Ho, S. J. Kim, and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 118–130.

[27] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 117–128.

[28] T. J. Ham, J. L. Aragón, and M. Martonosi, "Desc: Decoupled supply-compute communication management for heterogeneous architectures," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 191–203.

[29] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[30] N. Neves, P. Tomás, and N. Roma, "Compiler-assisted data streaming for regular code structures," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 483–494, 2020.

[31] R. Kruppe, J. Oppermann, L. Sommer, and A. Koch, "Extending llvm for lightweight spmd vectorization: Using simd and vector instructions easily from any language," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 278–279.

[32] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "Vw-slp: Auto-vectorization with adaptive vector width," ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243189

[33] A. Mishra, A. M. Malik, and B. Chapman, "Extending the llvm/clang framework for openmp metadirective support," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, 2020, pp. 33–44.

[34] S. Memeti and S. Pllana, "Hstream: A directive-based language extension for heterogeneous stream computing," in *2018 IEEE International Conference on Computational Science and Engineering (CSE)*, 2018, pp. 138–145.

[35] A. E. Şuşu, "A vector-length agnostic compiler for the connex-s accelerator with scratchpad memory," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 6, Oct. 2020. [Online]. Available: https://doi.org/10.1145/3406536

[36] LLVM, "LLVM Programmer's Manual." [Online]. Available: https://llvm.org/docs/ProgrammersManual.html

[37] ——, "Extending LLVM: Adding instructions, intrinsics, types, etc." [Online]. Available: https://llvm.org/docs/ExtendingLLVM.html

[38] ——, "The LLVM Target-Independent Code Generator." [Online]. Available: https://llvm.org/docs/CodeGenerator.html#live-intervals

[39] ——, "Writing an LLVM Backend." [Online]. Available: https://llvm.org/docs/WritingAnLLVMBackend.html#calling-conventions

[40] ——, "LLVM SubtargetFeatures Class Reference," https://llvm.org/doxygen/classllvm_1_1SubtargetFeatures.html.

[41] G. Hunter, "Supporting ARM's SVE in LLVM," https://lists.llvm.org/pipermail/llvm-dev/2016-November/106819.html.

[42] "Defining Fixups and Relocations," https://www.embecosm.com/appnotes/ean10/html/ch06s02.html.