TECHNISCHE UNIVERSITÄT DARMSTADT

Automatic Compiler Support for Application-Specific Instruction Set Architecture Extensions

Masterthesis by

Michael Halkenhäuser

February 2022

Examiner:
Prof. Dr. Andreas Koch

Supervisor:
Dr.-Ing. Julian Oppermann

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

Automatic Compiler Support for Application-Specific Instruction Set Architecture Extensions
Automatische Übersetzerunterstützung für anwendungsspezifische Befehlssatzerweiterungen
Masterthesis by Michael Halkenhäuser
Submitted on 13.02.2022
Examiner: Prof. Dr. Andreas Koch
Supervisor: Dr.-Ing. Julian Oppermann

Thesis Statement

I herewith formally declare that I, Michael Halkenhäuser, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Darmstadt, 13. February 2022

(Michael Halkenhäuser)

Acknowledgements

To my family and friends, a sincere *thank you* for the continued support and provided encouragement. This work would probably not have happened without these people.

Abstract

High demand for computational power over the last decades has led to the widespread presence of processors in our everyday lives. Simultaneous growth in the number of application areas requires specialized hardware, tailored for specific tasks. These *application-specific processors* are of key importance, since their limitation to a few possible operations offers distinct advantages. Such hardware-accelerators are generally able to outperform any common processor with regard to some metrics. The most frequent ones being computational speed or power consumption.

With availability of the RISC-V [6] open-standard processor architecture, development of customized processors becomes even more interesting. It allows the creation and implementation of instructions tailored to one's own prerequisites, generally without requiring licensing fees. Once these instructions are defined and an actual processor implementation is available, developers are faced with the lack of proper support by compilers. This in turn complicates the instructions' utilization considerably, requiring distinct knowledge of their actual binary representation or manual implementation of support [3].

Both options are time consuming and demanding with regard to the assumed proficiency. Therefore, we want to suggest an automated approach to the implementation of compiler support for these Instruction Set Architecture (ISA) eXtensions (ISAXs). Our presented infrastructure will be utilizing and operating on LLVM, an open-source compiler framework and automatically add support for certain instruction types. In this work, we will provide an introduction to automated source code customization with regard to the LLVM sub-project `LibTooling` and present the performed modifications in detail. Furthermore, we will introduce an input format based on LLVM's Intermediate Representation (IR) and show how certain instruction properties may be automatically determined. The last step of our approach will present the application of this information in order to ultimately enable compiler support. Afterwards, we will evaluate our approach with regard to reduced development effort and requirements. Lastly, we will summarize our work and conclude with the presentation of possible next steps with regard to the extension of our presented infrastructure.

Kurzfassung

Der hohe Bedarf an Rechenleistung hat in den letzten Jahrzehnten dazu geführt, dass Prozessoren in unserem Alltag sehr weit verbreitet sind. Die gleichzeitig wachsende Zahl von Anwendungsbereichen erfordert spezialisierte Hardware. Diese *anwendungsspezifischen Prozessoren* sind von zentraler Bedeutung, da ihre Beschränkung auf wenige, spezielle Operationen deutliche Vorteile mit sich bringt. Solche Hardware-Beschleuniger sind generell in der Lage, gewöhnliche Prozessoren in Bezug auf bestimmte Metriken zu übertreffen. Am häufigsten sind dies die Rechengeschwindigkeit oder der Stromverbrauch.

Mit der Verfügbarkeit des offenen Prozessorarchitektur-Standards RISC-V [6] wird die Entwicklung derartiger Prozessoren noch interessanter. RISC-V ermöglicht die Erstellung und Implementierung von Anweisungen, die auf die eigenen Voraussetzungen zugeschnitten sind, ohne an Lizenzgebühren gebunden zu sein. Sobald die gewünschten Anweisungen definiert sind und ein tatsächlicher Prozessor zur Verfügung steht, sehen sich Entwickler jedoch mit einem Mangel an geeigneter Compilerunterstützung konfrontiert. Dies wiederum erschwert die Nutzung der Befehle erheblich und erfordert Kenntnis ihrer Binärdarstellungen oder eine manuelle Implementierung der Unterstützung [3].

Beide Optionen sind zeitaufwendig und anspruchsvoll in Bezug auf die vorausgesetzten Kenntnisse. Daher möchten wir einen automatisierten Ansatz für die Implementierung der Compilerunterstützung für diese Erweiterungen (ISAX) vorschlagen. Die von uns vorgestellte Infrastruktur nutzt und arbeitet mit LLVM, einem Open-Source-Compiler-Framework, und fügt automatisch Unterstützung für bestimmte Befehlstypen hinzu. In dieser Arbeit werden wir eine Einführung in die automatisierte Quellcode-Anpassung im Hinblick auf das LLVM-Teilprojekt `LibTooling` geben und durchgeführte Änderungen im Detail vorstellen. Zudem stellen wir ein Eingabeformat vor, das auf LLVM's IR basiert und zeigen, wie bestimmte Anweisungseigenschaften automatisch bestimmt werden können. Der letzte Schritt unseres Ansatzes wird die Anwendung dieser Informationen präsentieren, um letztendlich neue Befehle zu unterstützen. Anschließend werden wir unseren Ansatz im Hinblick auf reduzierten Entwicklungsaufwand und verringerte Anforderungen bewerten. Letztlich fassen wir unsere Arbeit zusammen und schließen mit der Vorstellung möglicher Schritte im Hinblick auf die Erweiterung der von uns vorgestellten Infrastruktur ab.

Contents

1. Introduction

Since coprocessors became available in desktop computers during the 1970s, specialized hardware has become an integral part of computing. Perhaps the most prominent implementations of such application-specific architectures would be dedicated Graphics Processing Units (GPUs) or motherboard chipsets. Each of these can be perceived as a hardware accelerator, primarily designed with a particular task in mind. And while this task could generally also be fulfilled by a Central Processing Unit (CPU), specialized hardware is usually far more efficient with regard to certain requirements. Examples would be increased *speed*, enhanced *parallelism*, improved *utilization* and lowered *power consumption* among others. Basic adjustments to meet those requirements using CPUs are for example: increased clock rate (speed) or improved manufacturing process (power).

However, physical limitations prohibit arbitrary performance gains and must be taken into consideration. Raised clock rates for example will result in higher power usage and in turn more dissipated heat, where doubled power consumption yields considerably less than double performance. Another way to improve the efficiency of CPUs is the extension of their ISA for frequent tasks, basically enabling hardware acceleration within the CPU. Some of these ISAXs have become very popular, like vector (e.g. SSE, AVX) and cryptography (en-/decryption using AES) related instruction sets.

Many specialized niche tasks, on the other hand, will most likely never get widespread ISA support. Nevertheless, the amount of gathered data has increased in nearly every area and digital infrastructures continue to develop. Thus, the demand for domain specific computations has grown as well. Here, industrial facilities and applications can be taken as an example. Continuously collecting and calculating performance or maintenance indicators may depend on narrow timing or require low power consumption in remote areas.

ISAX development generally involves proprietary licenses, trademarks and fees to use or customize, which significantly raises the entry threshold. One such example would be the ARM architecture, where a customer may use intellectual properties in exchange for per-product fees. RISC-V [6] in contrast is an *open standard ISA*, designed for practical use and provided under open source licenses. This in turn makes RISC-V very interesting and suitable for a wide variety of applications, from small embedded systems to personal or even supercomputers.

Since instructions of a non-standard RISC-V ISAX are unknown to compilers, there will be no appropriate support. Therefore, it is generally impossible to utilize these instructions other than directly using their corresponding binary opcodes. Not only does this process require a considerable amount of time, it is also quite error-prone. While implementing compiler support is a one-time effort with regard to ratified extensions, each time an ISAX changes, the compiler has to be adapted. Thus, it would be desirable to keep compiler support, even during the development of a custom extension.

In the course of this work we will present a *tool* which automates the implementation of support for certain given RISC-V ISAX in a LLVM compiler framework [2]. LLVM is particularly interesting because of its open-source nature, active community and modularity. One of its core features is the LLVM Intermediate Representation (LLVM-IR), which is roughly comparable to assembler code. Program analyses and transformations in LLVM are designed to run on LLVM-IR and are therefore language agnostic. Because of its versatility LLVM-IR allows for incorporation of custom data, which can be retrieved and manipulated using LLVM's infrastructure.

Our presented tool receives the ISAX as a (human-readable) LLVM-IR file and generates patches for the corresponding LLVM source files. After application of these patches and recompilation, the LLVM project features C++ support for the newly defined instructions in the form of callable *intrinsic functions* (Chapter 2). Currently, we are able to support automatic determination of some instruction properties (Section 3.2) and offer a platform which may be extended by more instruction or data types and other advanced features (Chapter 6). While we were working primarily towards ISAX support, the created infrastructure also allows to make changes in general, especially in C/C++ based projects (using `LibTooling`).

As a simplified example, we will take a look at an ISAX comprised of merely one *multiply-accumulate* instruction (Listing 1).

```
1 ; Omitted for simplicity (among other things):
2 ; register semantics and corresponding loads / stores
3 define i32 @mac(i32 %a, i32 %b, i32 %c)
4 {
5         entry:
6           %tmp = mul i32 %b, %c
7           %a = add i32 %tmp, %a
8           ret %a
9 }
```

Listing 1: MAC pseudo-LLVM-IR definition

Assuming there was no compiler support for this new instruction, it could still be used in C++. This can be achieved by writing *inline-assembler* code (Listing 2). But, since mnemonics are part of the compiler support, one has to provide the bit pattern directly.

```
// Note: out- / inputs & clobbered registers
//       also have to be set manually [here]
#ifdef MAC_SUPPORT_AVAILABLE
// Assembler mnemonics require compiler support
asm volatile ("mac a2, a1, a0" : [here]);                // (1)
#else
// If no support is available, mandatory use of the bit pattern
asm volatile (".byte 0x33, 0x86, 0xA5, 0x42" : [here]);  // (2)
#endif
```

Listing 2: C++ sample use of inline assembler

Patch files, created by our tool, will define a corresponding C++ intrinsic (Listing 3). Using this function in a source file provides direct access to the new instruction and requires no further work by the programmer.

```
int a = 1, b = 2, c = 3;
__builtin_riscv_mac(a, b, c); // a will hold value  '7'
__builtin_riscv_mac(a, b, c); // a will hold value '13'
```

Listing 3: C++ sample use of an intrinsic

In the next chapter, we will take a look at LLVM's layers with special regard to ISAX support and source code modifications. Chapter 3 presents our approach to realize and apply these changes utilizing the user-provided information and automatically deduced instruction properties. Related work will be discussed in Chapter 4, leading to an evaluation of our infrastructure in Chapter 5. Finally, conclusions are drawn in Chapter 6, where we will also discuss possible extensions to this work.

2. Background

In this chapter relevant changes to LLVM[1] are discussed, in order to support new instructions. We will begin with an abstract overview and then split the examination into source code and *TableGen*[2] changes. Along the way we will provide short introductions to TableGen and `libtooling` with regard to our work.

Furthermore, we want to establish a small custom ISAX named `DSP`, including a single new instruction `mac` (cf. Listing 3). If applicable, we will illustrate certain changes with this example. After modification and rebuilding of the target LLVM, a developer will be able to use the corresponding C++ intrinsic (named `__builtin_DSP_mac`). Consider the example[3] invocation of `clang++`[4] in Listing 4 as outlook. It depicts a possible way for manual review of C++ and LLVM intrinsic coupling. Here, we are looking at a theoretical target processor with the following set of supported *features*, i.e. the entirety of all ISAXs implemented by it:

- `RV32I` – Base Integer Instruction Set, 32-bit
- `M` – Standard Extension for Integer Multiplication and Division
- `DSP` – Non-standard user-level extension 'DSP', version 0.1

```
clang++ --target=riscv32 \  # (Modified clang++)
  # Set architecture features / extensions (base: rv32i, m):
  #  + a custom ISAX: kind (x), name (DSP) and version (0p1)
    -march=rv32imxDSP0p1 \
    -menable-experimental-extensions \
  # Input CPP file, calling C++ intrinsic '__builtin_riscv_DSP_mac'
    mac_test.cpp -S -emit-llvm -o mac_test.ll
  # Output LLVM-IR file will use LLVM intrinsic '@llvm.riscv.DSP.mac'
```

Listing 4: Sample `clang++` call (C++ to LLVM intrinsics)

[1] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-13.0.0`
[2] `https://llvm.org/docs/TableGen/`
[3] For a more sophisticated example (e.g. with assembler output) refer to Listing 32
[4] `https://clang.llvm.org/`

2.1. Modification Overview

Regarding support of new instructions, we will follow LLVM's pipeline (Figure 2.1) and consequently start at its C language family frontend `clang`. In order to allow the utilization of a custom ISAX when calling `clang`, we need to define a new, uniquely named extension. Additionally, intrinsics (also called: *builtin functions*) have to be defined which may then be used within the provided source code.

To allow a mapping of these function calls to actual processor instructions, we will use LLVM intrinsics as an intermediate step. Therefore, the C++ builtin function names have to be coupled with their LLVM counterpart using TableGen files on the LLVM-IR level. Then LLVM intrinsics are linked to their actual instruction definitions at the middle- and backend of the pipeline, by extensive use of generated records. Finally, along with this information, we have to define architecture dependent registers and the extension to allow the selection of these instructions.

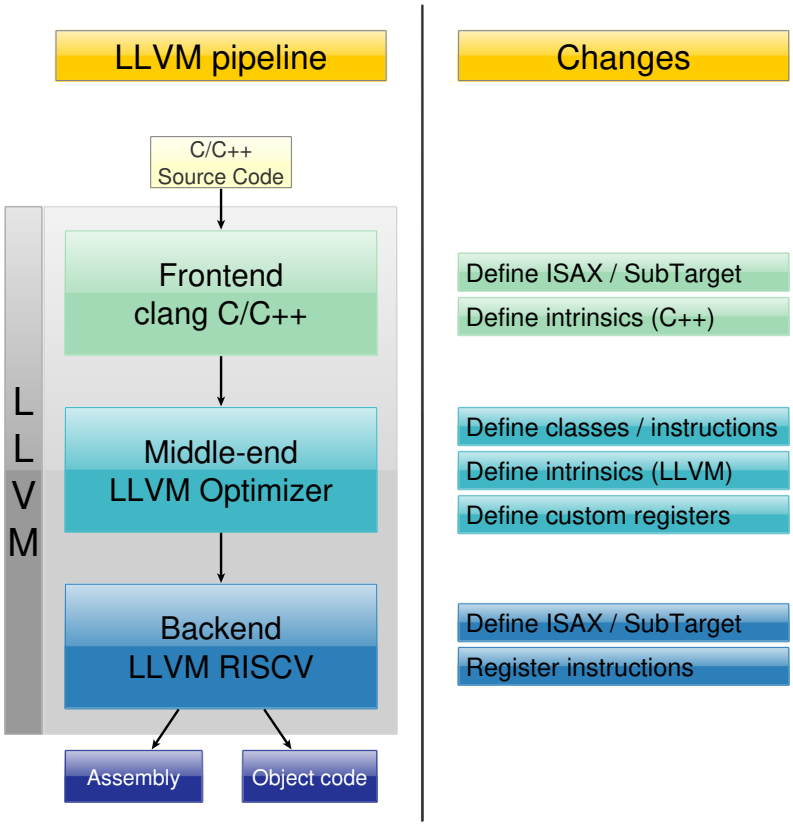

Figure 2.1.: Abstract Modification Overview

2.2. Source Modifications

This section will present the necessary changes to LLVM's source files (`.h`/`.cpp`) in detail. Each subsection will be named after the file's relative path and describe the modifications as well as their effect.

2.2.1 `./clang/include/clang/Basic/BuiltinsRISCV.def`

In order to recognize the C++ intrinsics in the frontend, they e.g. have to be defined as *builtins* in this file. Therefore, we have to add a builtin function definition using the `TARGET_BUILTIN`macro for each ISAX instruction. Hence, the function's name and signature, as expected within C++ sources as well as attributes and the required feature's name have to be provided.

```
// For more info, see also "./clang/include/clang/Basic/Builtins.def"
// Macro:  define TARGET_BUILTIN(ID, TYPE, ATTRS, FEATURE)
//                        BUILTIN(ID, TYPE, ATTRS)
TARGET_BUILTIN(__builtin_dsp_mac, "iiii", "nc", "experimental-xdsp")
// [...]
#undef BUILTIN
#undef TARGET_BUILTIN
```

Listing 5: Modifications of `./clang/include/clang/Basic/BuiltinsRISCV.def`

2.2.2 `./clang/lib/Basic/Targets/RISCV.h`

The frontend has to be able to store / query each present / possible feature from a targeted processor. `TargetInfo` manages this information, hence we have to add a member for our ISAX in the RISC-V specific class (Listing 6).

```
// RISC-V Target
class RISCVTargetInfo : public TargetInfo {
protected:
  // Add member to query / store feature support
  bool HasDSP = false;
// [...]
}
```

Listing 6: Modifications of `./clang/lib/Basic/Targets/RISCV.h`

2.2.3 ./clang/lib/Basic/Targets/RISCV.cpp

This frontend file is very important and requires three changes (Listing 7), which may set and query the member defined in Listing 6. Thus, if we enable the feature `experimental-xDSP` the member `HasDSP` has to be set to `true`, and we define a macro that may be used internally (to conform with other experimental features).

```
1  // Change (1)
2  void RISCVTargetInfo::getTargetDefines( /* ... */ ) {
3    // [].../]
4    if (HasDSP)
5      Builder.defineMacro("__riscv_dsp", "10000");
6  }
7  // [...]
8  // Change (2)
9  bool RISCVTargetInfo::hasFeature(StringRef Feature) const {
10   bool Is64Bit = getTriple().getArch() == llvm::Triple::riscv64;
11   return llvm::StringSwitch<bool>(Feature)
12   .Case("riscv", true)
13   // [...]
14   .Case("experimental-xdsp", HasDSP)
15   .Default(false);
16 }
17 // [...]
18 // Change (3)
19 bool RISCVTargetInfo::handleTargetFeatures( /* ... */ ) {
20   for (const auto &Feature : Features) {
21     // [...]
22     else if (Feature == "+experimental-xdsp")
23       HasDSP = true;
24   }
25   return true;
26 }
```

Listing 7: Modifications of `./clang/lib/Basic/Targets/RISCV.cpp`

2.2.4 ./clang/lib/Driver/ToolChains/Arch/RISCV.cpp

This frontend related file handles target features, which may be queried from the outside, e.g. using "getRISCVTargetFeatures". Our goal is to add a valid return (RISCVExtensionVersion) in "isExperimentalExtension" when the ISAX is requested by a call to clang (e.g. using -march=rv32i**xdsp0p2**). Hence, we have to add another conditional return as shown in Listing 8 (lines 5+7).

```
1 /* ... */ isExperimentalExtension(StringRef Ext) {
2   // [...]
3   // {prefix = "x"}{feature-name = "dsp"}{suffix = ""} --> "xdsp"
4   if(Ext == "xdsp")
5     // Return major ("0") and minor ("2") version of the feature
6     return RISCVExtensionVersion{ "0", "2" };
7   return None;
8 }
```

Listing 8: Modifications of ./clang/lib/Driver/ToolChains/Arch/RISCV.cpp

2.2.5 ./llvm/lib/Target/RISCV/RISCVSubtarget.h

Connecting front- and backend-end this file will provide the means to allow the coupling of a TableGen definition (FeatureDSP, see Section 2.4) and a way to query available target features (hasNonStdExtDSP()). Therefore, we have to define a new member and corresponding getter (Listing 9).

```
1 class RISCVSubtarget : public RISCVGenSubtargetInfo {
2   bool HasNonStdExtDSP = false;                    // Change (1)
3   bool HasStdExtM = false;
4   // [...]
5 public:
6   bool hasNonStdExtDSP() const { return HasNonStdExtDSP; }  // (2)
7   bool hasStdExtM() const { return HasStdExtM; }
8   // [...]
9 }
```

Listing 9: Modifications of ./llvm/lib/Target/RISCV/RISCVSubtarget.h

2.2.6 ./llvm/lib/Target/RISCV/Disassembler/RISCVDisassembler.cpp

This backend file is part of our *experimental register support* and required by LLVM if a register class was defined in the corresponding TableGen file (Section 2.4). We assume that the register class was named `FPR42` and offers 8 register fields. Furthermore, we defined one actual instance of this register named `FPR42_REG`. Then the required addition would look like Listing 10.

```
// Define: "Decode< Register class name >RegisterClass"
static DecodeStatus DecodeFPR42RegisterClass(MCInst &Inst,
         uint64_t RegNo, uint64_t Address, const void *Decoder) {
  // Size information is used to prevent out-of-bounds access
  if (RegNo >= 8) { return MCDisassembler::Fail; }
  // Usage of actual register instance
  MCRegister Reg = RISCV::FPR42_REG_0 + RegNo;
  Inst.addOperand(MCOperand::createReg(Reg));
  return MCDisassembler::Success;
}
```

Listing 10: Modifications of `./llvm/lib/Target/RISCV/Disassembler/RISCVDisassembler.cpp`

2.3. TableGen Syntax

Throughout the LLVM build process, TableGen is used to generate domain-specific information. Its syntax was designed to reduce maintenance effort for developers and to offer high flexibility along with good scalability. Generated TableGen output is usually stored in `".inc"` files within LLVM's *build directory* which often contain generated C++ source code. Input data on the other hand is generally stored in so-called *target-definition* files (`".td"`, inside the *source directory*) and composed of merely three key concepts. The most basic one being `records`, which in turn are divided into `classes` and `definitions`. TableGen `records` have a unique identifier, a list of parameters and classes from which they inherit. A `record` instance, without undefined values is considered a `definition`. Lastly, TableGen `classes` are abstract records, which may have parameters and can be used to form more complex, composite records.

Similar to the C++ `#include` pragma, it is possible to combine multiple files and further reduce redundancy. For example Listing 11 will create the definition "FeatureISAX" of class `SubtargetFeature`, which was defined in another file.

```
include "llvm/Target/Target.td"
def FeatureISAX : SubtargetFeature<"xisax", "HasISAX",
                    "true", "Enable custom ISAX">;
```

Listing 11: TableGen `SubtargetFeature` example

Listing 12 shows the slightly shortened `SubtargetFeature` class from LLVM.

```
// Defined in "./llvm/include/llvm/Target/Target.td"
class SubtargetFeature<string n, string a,  string v,
        string d, list<SubtargetFeature> i = []> {
  // Name - Feature name.  Used by (-mattr=)
  string Name = n;
  // Attribute - Attribute to be set by feature.
  string Attribute = a;
  // Value - Value the attribute to be set to.
  string Value = v;
  // Desc - Feature description.  Used by (-mattr=)
  string Desc = d;
  // Implies - Implied, present features.
  list<SubtargetFeature> Implies = i;
}
```

Listing 12: TableGen `SubtargetFeature` class example

In addition to these basic concepts, we want to cover two more advanced TableGen features, namely `let` statements and `multiclass` records. `let` statements allow the programmer to override certain field values. Records, defined within the scope of a let, will use the provided value instead of their default. This adds to the flexibility of all definitions, as they become easily reusable. `multiclass` records, on the other hand, improve on scalability as they allow for simultaneous instantiations of multiple definitions. Therefore, the developer only has to put several definition statements within the multiclass. When creating an instance via `defm` each internal definition will be instantiated.

All these features can be combined with loops (e.g. `foreach`) and other builtin types or functions to generate arbitrarily complex hierarchies. To conclude this introduction and illustrate possible interactions, we will provide a practical example (Listing 13). Its code implements the first modification of the next section: it couples each C++ intrinsic with their LLVM counterpart.

```
// Each record inside this scope will have
// the field "TargetPrefix" set to "riscv"
let TargetPrefix = "riscv" in {
  multiclass ISAXIntrinsic<
    list<LLVMType> ret_types,
    list<LLVMType> param_types,
    list<IntrinsicProperty> intr_properties> {
      // Multiple inheritance
      def int_riscv_isax_ # NAME
        : GCCBuiltin<"__builtin_isax_" # NAME>,
          Intrinsic<ret_types,param_types,intr_properties>;

      // Constructed, additional def
      // Always create a second, void return-type intrinsic
      def int_riscv_isax_void_ # NAME
        : GCCBuiltin<"__builtin_isax_" # NAME>,
          Intrinsic<       [],param_types,intr_properties>;
  }

  // This defm invocation will create two definitions:
  //   int_riscv_isax_mac_int
  //   int_riscv_isax_void_mac_int
  defm mac_int : ISAXIntrinsic<
                   [llvm_i32_ty],
                   [llvm_i32_ty,llvm_i32_ty,llvm_i32_ty],
                   [IntrReadMem]>;
} // TargetPrefix = "riscv"
```

Listing 13: TableGen `multiclass` example

2.4. TableGen Modifications

This section will present the necessary changes to LLVM's TableGen files (`.td`) in detail. Each subsection will be named after the file's relative path and describe the modifications as well as their effect.

2.4.1 `./llvm/include/llvm/IR/IntrinsicsRISCV.td`

This file connects front- and middle-end by coupling C++ intrinsics with their LLVM counterparts. To achieve this, we have to define a class that takes signature and properties as well as the default TableGen field `"NAME"` to instantiate an accordingly named LLVM intrinsic (Listing 14).

```
let TargetPrefix = "riscv" in {

  multiclass DSPIntrinsic<list<LLVMType> ret_types,
              list<LLVMType> param_types,
              list<IntrinsicProperty> intr_properties> {
  def int_riscv_dsp_ # NAME
      : GCCBuiltin<"__builtin_dsp_" # NAME>,
        Intrinsic<ret_types,param_types,intr_properties>;
  }
  // For more info, see also:
  // "./llvm/include/llvm/IR/Intrinsics.td"
  // e.g. definition of type "llvm_i32_ty": line 242
  defm mac : DSPIntrinsic<[llvm_i32_ty],
    [llvm_i32_ty,llvm_i32_ty,llvm_i32_ty],[IntrNoMem]>;
  // ... remaining intrinsic definitions

} // TargetPrefix = "riscv"
```

Listing 14: Modifications of `./llvm/include/llvm/IR/IntrinsicsRISCV.td`

2.4.2 ./llvm/lib/Target/RISCV/RISCVInstrInfo.td

Primarily connecting middle- and backend, this file is the most important with regard to actual processor instructions. But it also defines the feature query of our ISAX (Listing 15), which transfers knowledge from front- to the backend (see Section 2.2). All instruction encodings, assembler representations and the coupling of LLVM intrinsics to instructions are placed here, which is why we would advise to create a separate file and include it here. Additionally, since we want to support more instruction formats in the future, we will define a custom instruction class (Listing 36) from which all our definitions will inherit. Instruction definition examples using this *super-class* are given in Listing 16, it presents the actual, derived classes and how they are instantiated. Afterwards, they may be used to create a pattern, which allows replacing the LLVM intrinsic by its corresponding ISAX instruction.

```
1  // Note: This could also e.g. be placed into
2  // "./llvm/lib/Target/RISCV/RISCV.td"
3  // (Usually, "SubtargetFeatures" are defined there.)
4  def FeatureDSP :
5        SubtargetFeature<"experimental-xdsp",
6                         "HasNonStdExtDSP", "true",
7                         "''DSP'' (non-standard extension)">;
8  def HasNonStdExtDSP :
9        Predicate<"Subtarget->hasNonStdExtDSP()">,
10                 AssemblerPredicate<(all_of FeatureDSP),
11                 "DSP Instruction Set">;
```

Listing 15: Modifications of `./llvm/lib/Target/RISCV/RISCVInstrInfo.td`

```
// (1) Instruction class definitions
let Predicates = [HasNonStdExtDSP] in {
  let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
  class DSP_mac
        : RVDSPISAXInst<(outs GPR:$res0),
                        (ins GPR:$rd, GPR:$rs1, GPR:$rs2),
                        "dsp.mac", "$res0, $rd, $rs1, $rs2",
                        InstFormatR>,
        Sched<[WriteIALU, ReadIALU, ReadIALU]> {
    bits<5> rd;
    bits<5> rs1;
    bits<5> rs2;
    // Encoding was provided in MetaData node
    // "!isax.func.inst.def.encoding" =
    // "7'b0100001 rs2[4:0] rs1[4:0] 3'b000 rd[4:0] 7'b0110011"
    let Inst{31-25} = 0b0100001;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000;
    let Inst{11-7}  = rd;
    let Inst{6-0}   = 0b0110011;
  }
  // [...] more classes -- one for each instruction
} // Predicates = [HasNonStdExtDSP]

// (2) Define instance of 'DSP_mac' class
def mac : DSP_mac;
// [...] each instruction class is instantiated

// (3) Intrinsic replacement patterns
let Predicates = [HasNonStdExtDSP] in {

def : Pat<(int_riscv_dsp_mac GPR:$rd, GPR:$rs1, GPR:$rs2),
          (mac_int_1  GPR:$rd, GPR:$rs1, GPR:$rs2)>;

// [...] more patterns (map instructions to LLVM intrinsic counterparts)

} // Predicates = [HasNonStdExtDSP]
```

Listing 16: Modifications of `./llvm/lib/Target/RISCV/RISCVInstrInfo.td`

2.4.3 ./llvm/lib/Target/RISCV/RISCVRegisterInfo.td

Modifying this file is part of our *experimental register support*, located in the backend of LLVM. Here, we will add a register class, instantiate each actual register "DSPREG" and their elements. As an example (Listing 17), we will define "DSPREG" of class "DSPR", offering two register fields.

```
// Actual register class instance
let RegAltNameIndices = [ABIRegAltName] in {
  // Define each register field with an ID
  // and potentially an alternative name
  def DSPREG0 : RISCVReg<0, "dspreg0", ["dspreg0"]>, DwarfRegNum<[-1]>;
  def DSPREG1 : RISCVReg<1, "dspreg1", ["dspreg1"]>, DwarfRegNum<[-1]>;
}

// Register class
def DSPR : RegisterClass<"RISCV", [i32], 32, (add
  (sequence "DSPREG%u", 0, 1)
)>;
```

Listing 17: Modifications of `./llvm/lib/Target/RISCV/RISCVRegisterInfo.td`

2.5. C/C++ source code matching: LibTooling & LibASTMatchers

In this section, we want to provide the necessary information on how to automatically identify C/C++ source code positions. To accomplish this task, `LibTooling` and `LibASTMatchers` (both part of `clang-tools-extra`) are very important. Together, they offer the functionality to parse and match C++ source code on Abstract Syntax Tree (AST)-level.

Since a full-fledged introduction[5] would go beyond the scope of this work, we will omit e.g. the project setup and detailed descriptions of AST matchers[6]. Instead, we want to start with a very simple C program and illustrate how to obtain its AST by using `clang`. This will lead to our first, simple AST matcher for `clang-query`, a standalone tool able to retrieve matched results. Afterwards, we will use this information in `LibTooling` context and sketch a possible result examination.

```
1 int main() {
2   int i = 42;
3   return (i) ? 0 : 1;
4 }
```

Listing 18: AST sample program `"ex_ast.c"`

Assuming our goal was to identify certain positions in Listing 18: (1) end of integer definition `"i"` and (2) the condition `(i)` within the `return` statement. Of course, we could use text-based search in such a simple case, or if we had no means (tool support) of parsing the input like when dealing with TableGen files. However, programs may become very complex and AST-based matching can offer superior robustness as well as an increased level of abstraction. Our first step towards the AST representation of `clang` will be to actually generate it from the example (Listing 19).

[5] Please, refer to: `https://clang.llvm.org/docs/LibASTMatchersTutorial.html`

[6] Please, refer to: `https://clang.llvm.org/docs/LibASTMatchersReference.html`

```
# Pipe program into clang (note: trailing dash) and dump its AST
# '-Xclang':       Pass the following argument directly to the
#                  clang compiler (i.e., not front-end)
# '-ast-dump':     Dump the program's AST
# '-fsyntax-only': Syntax check but no compilation
# '-x':            Input language (here: 'c', can also be 'c++')
cat ex_ast.c | clang -Xclang -ast-dump -fsyntax-only -x c -
# Note: when exporting an AST to file it may be useful
#       to suppress color codes by using:
#       -fno-color-diagnostics
# ----------------------------
# Identical command, but without pipe:
clang -Xclang -ast-dump -fsyntax-only -x c ex_ast.c
```

Listing 19: Using `clang` for an AST dump

The command(s) in Listing 19 will produce similar output to Figure 2.2, which was slightly shortened. Certain code portions may be recognized quite easily by *reading* the AST, like `main`'s `FunctionDecl` or `VarDecl` of `i` (part of position (1)). Others may be harder to spot, because of their indentation levels or unfamiliar notations, like `ParenExpr` which represents our designated position (2).

```
TranslationUnitDecl 0x1179978 <<invalid sloc>> <invalid sloc>
|-
    [...]
|-
`-FunctionDecl 0x11bba50 <<stdin>:1:1, line:4:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x11bbcc8 <col:12, line:4:1>
    |-DeclStmt 0x11bbbd8 <line:2:3, col:13>
    | `-VarDecl 0x11bbb50 <col:3, col:11> col:7 used i 'int' cinit
    |   `-IntegerLiteral 0x11bbbb8 <col:11> 'int' 42
    `-ReturnStmt 0x11bbcb8 <line:3:3, col:20>
      `-ConditionalOperator 0x11bbc88 <col:10, col:20> 'int'
        |-ImplicitCastExpr 0x11bbc70 <col:10, col:12> 'int' <LValueToRValue>
        | `-ParenExpr 0x11bbc10 <col:10, col:12> 'int' lvalue
        |   `-DeclRefExpr 0x11bbbf0 <col:11> 'int' lvalue Var 0x11bbb50 'i' 'int'
        |-IntegerLiteral 0x11bbc30 <col:16> 'int' 0
        `-IntegerLiteral 0x11bbc50 <col:20> 'int' 1
```

Figure 2.2.: AST of `ex_ast.c` (Listing 18)

To actually start writing AST matchers we can use `clang-query`, which allows interactively testing, debugging and visualizing results. Listing 20 shows how the example can be loaded.

```
# Load program 'ex_ast.c' into clang-query
# Note: two trailing dashes (omit 'compilation database')
# '--extra-arg': Append the following argument to the
#                implicitly used compiler
clang-query ex_ast.c --extra-arg="-x" --extra-arg="c" --
```

Listing 20: `clang-query` invocation

Listing 21 presents a collection of queries and their effect (again, the official introduction and reference can be found here).

Along with that we will use so-called *bindings*, which let us name certain matched nodes of the AST. This feature will be helpful to manage and retrieve our target positions. To disable default behavior (each node will receive a binding called ”root”), use the command `"set bind-root false"`.

The last matcher example from Listing 21 is advanced, but can be *read* like this:

- Match a `functionDecl` only present in our source file, with name ”main” and a `compoundStmt` descendant (curly braces)
- This `compoundStmt`, must have two descendants:
  - A `declStmt` which holds the `varDecl` of `"i"`
  - A `returnStmt` with a pair of parentheses

```
// Match all declarations (can also be shortened to "m decl()")
match decl()
// Result: 8 matches (6 are -NOT- part of the actual source!)
//===----------------------------------------------------------------------===//
// Only match declarations which are in the actual source file
// Note: Useful when many files are #include'd
m decl(isExpansionInMainFile())
// Result: 2 matches -- "main" and "i"
//===----------------------------------------------------------------------===//
// Match (named / function) decl with name "main"
m decl(allOf(isExpansionInMainFile(),namedDecl(hasName("main"))))
m functionDecl(allOf(isExpansionInMainFile(),hasName("main")))
  .bind("func_main")
// Result: 1 match -- "main" with binding "func_main"
//===----------------------------------------------------------------------===//
// After some experimenting and looking at the AST we could write ...
// Note: to issue this command -- format as a single line
// (Explanation is given in text)
m functionDecl(allOf(
    isExpansionInMainFile(),
    hasName("main"),
    hasDescendant(compoundStmt(allOf(
      hasDescendant(declStmt(
        hasDescendant(varDecl(hasName("i")))).bind("decl")),
      hasDescendant(returnStmt(
        hasDescendant(parenExpr().bind("cond_parentheses"))
      ).bind("returnStmt")))
    ))
  )).bind("func_main")
```

Listing 21: `clang-query` AST-matcher examples

In a LibTooling project we can define and run() this matcher as shown in Listing 22.

```
// Note: Includes and namespaces omitted for readability
// Define our own, custom OptionCategory
static llvm::cl::OptionCategory EX_TOOL("LibTooling Example");

int main(int argc, const char** argv) {
  // Parse command line arguments
  auto OPOrErr = CommonOptionsParser::create(argc, argv, EX_TOOL);
  if (auto E = OPOrErr.takeError()) {
    // May print error message using toString(std::move(E))
    return 1;
  }
  auto& OP = *OPOrErr;
  RefactoringTool Tool(OP.getCompilations(), OP.getSourcePathList());

  DeclarationMatcher exampleMatcher =
    /* Insert exact matcher used in clang-query (without 'm') */ ;
  PositionFinderCallBack exampleCallback; // Construct CallBack-class

  // This is the actual point of matcher registration and reason to
  // inherit from 'clang::ast_matchers::MatchFinder::MatchCallback'
  MatchFinder Finder;
  Finder.addMatcher(exampleMatcher, exampleCallback);

  // Create an action, coupled with the MatchFinder
  auto action = newFrontendActionFactory(&Finder);
  // Run the Tool using exampleMatcher -- in case of a match:
  //   the callback's run() method will be executed.
  return Tool.run(action.get());
}

class PositionFinderCallBack : public MatchFinder::MatchCallback {
  void run(const MatchFinder::MatchResult& Result) override {
    // Implement handling of an actual result here
  }
}
```

Listing 22: LibTooling usage example

The general concept in the usage example of `LibTooling` (Listing 22) is that our *tool* becomes a `clang-tool`, e.g. with the advantage of native command-line option parsing. An invocation of our example tool would look similar to `"ex_-tool ex_ast.c --"`. Provided options will then be parsed, like the source filename `ex_ast` and can be retrieved as an object. Most important with regard to our goal are user-provided `Matchers` and `Callbacks`, which are coupled through a `MatchFinder`. Each time a matcher encounters a result on the current AST, the corresponding `Callback` (i.e. its `run()` method) will be executed. To achieve all these links we need to use the `newFrontendActionFactory` to produce an object that is executable by a `RefactoringTool`, the topmost object instance. Calling its `run` method will start the actual matching, which may take some time (up to several minutes, for `DEBUG` builds) when dealing with *large* source files.

The last step in order to identify our two target positions will be to actually implement the `run()` method of a `MatchCallback` derived class. Reminder: from Listing 18 we wanted to find the position directly behind the definition of `i` and both parentheses in the return statement. Listing 23 provides such an implementation example. Since our class's `run` method was called from a result, using the (final) matcher from Listing 21 we can be sure that each *binding* is available. Therefore, we can simply load these into the corresponding object type. Most of these types offer a way to retrieve a `SourceLocation`, which can be interpreted by the result's `SourceManager` and directly yield an actual byte-offset. Similarly, we are able to calculate source ranges, that enable precise modifications within a given source file.

Now, that important files and locations, a basic understanding of TableGen code and automatic examination of source files are available, we want to proceed with our approach.

```
// Note: Includes and namespaces omitted for readability
void run(const MatchFinder::MatchResult& Result) override {
  // Retrieve the corresponding source manager
  // Mandatory to get source file related information
  auto& SM = *Result.SourceManager;

  // Load mandatory binds
  // These binds are a precondition to entering this method
  auto decl_i =
    Result.Nodes.getNodeAs<DeclStmt>("decl");
  auto condParentheses =
    Result.Nodes.getNodeAs<ParenExpr>("cond_parentheses");

  // Get end position of "int i"
  // + offset 1 since we want the position behind the semicolon
  SourceLocation posDecl_i = decl_i->getEndLoc().getLocWithOffset(1);
  // Get positions of left and
  // right parentheses (+1: get pos. behind RParen)
  SourceLocation posLParen =
    condParentheses->getLParen();
  SourceLocation posRParen =
    condParentheses->getRParen().getLocWithOffset(1);

  // Calculate range, so we could
  // e.g. replace the content of the parentheses
  auto rangeParen =
    CharSourceRange::getTokenRange(posLParen, posRParen);
  // FileByteRange: Offers "Length", "FileOffset" and "FilePath"
  auto tmpRange = FileByteRange(SM, rangeParen);

  // Get the byte offsets of our target locations (source file)
  // Byte offsets may be hard to "read" but are generally more useful
  SM.getFileOffset(posDecl_i) // Position (1)   = 26
  SM.getFileOffset(posLParen) // Position (2.1) = 36
  SM.getFileOffset(posRParen) // Position (2.2) = 39

  // We could also store some data into class members
  // and let future method calls handle the result.
}
```

Listing 23: `MatchCallback::run()` implementation example

3. Approach

This chapter will discuss how we automated the process of generating *patches* for a given LLVM source tree (here: release version `llvmorg-13.0.0`). In order to realize the following steps of information collection and derivation, we will assume that the targeted LLVM was already built. To ensure that the required libraries and applications are available, we will provide a set of recommended build options.

For example, `LibTooling` (as discussed in Section 2.5) is very important and offers the functionality to parse and match C++ source code on AST-level. Furthermore, output patch files will comply to the format expected by `clang-apply-replacements`. This application will be used to ultimately modify LLVM's source files. After applying the generated patches, the targeted LLVM has to be built again. It can then be used to (cross-) compile C++ sources for RISC-V architectures offering the provided ISAX (cf. Listing 4).

3.1. Input format

Since we are working with and will be modifying LLVM, using LLVM-IR as format and storage option comes naturally. In addition to present infrastructure for loading and iterating actual LLVM module and function definitions, LLVM-IR offers a mechanism to store and manage data which does not affect program correctness: `MetaData`[1]. This will allow us to store information at different levels: from settings like the ISAX's name at the global scope to properties of a single instruction or variable definition (like register information). An example of these cases is shown in Listing 24, with metadata at global level (`isax.feature.name`), a global LLVM variable `@X [...] isRegister` (referring metadata node `"2"`) for a register definition and `isax.func.inst.def.format` for specific instruction information.

These metadata nodes can be retrieved quite easily and for the sake of completeness we will also present the code necessary to actually load the LLVM module in Listing 25.

[1] `https://llvm.org/docs/LangRef.html#named-metadata`

```
!isax.feature.name = !{!0}    ; !{!"DSP"}
!isax.feature.prefix = !{!1}  ; !{!"x"}
; More global ISAX info ...

; Sample representation of RISC-V GPR register 'X'
@X = global [32 x i32] zeroinitializer, !isRegister !2

; Sample instruction 'foo'
define void @foo(i5 %rd, i5 %rs1, i5 %rs2)
!isax.func.inst.def.format !{!"R"}
; More function related information ...
{
  entry:
  ; May be empty
  ; but can be used to derive further information
  ; (e.g. register usage)
  ret void
}

; More instruction definitions ...

; Note the indirection, when dealing with global MD!
; "element" has to be stored as anonymous MDNode first
!0 = !{!"DSP"}
!1 = !{!"x"}
!2 = !{!"standard", !"GPR"}
```

Listing 24: MetaData example `ex_isax.ll`

When dealing with metadata nodes, it is crucial to keep track of their identifiers, nesting levels and the expected data types. This is why we chose to (1) explicitly load[2] (i.e. by its exact identifier) and (2) only use (`MD-`)`String` typed metadata. Currently defined node identifiers will be presented in the next two sections (a metadata overview can be found here: Table A.2).

[2] Another possible way would be to collect and iterate present metadata of the LLVM module or contained function definitions.

```
// Omitted includes, namespaces, variable setup for readability

// Provide the target filename e.g. "ex_isax.ll"
const char* FileName;
// Context object to manage Module information
// Note: consider "global" construction, since it
//       will be destroyed when leaving the scope
//       (may then lead to sporadic SIGSEGV)
llvm::LLVMContext CTX;
// Diagnostic object, may be used for error handling
SMDiagnostic Err;

// Load the LLVM module from file (handle like: Module* MOD)
std::shared_ptr<Module> MOD(parseIRFile(FileName, Err, CTX));
// Handle errors -- Check Err.getKind() and/or (MOD == nullptr)

// Prepare a string variable to hold the metadata
std::string featName;
// Not necessary in this case, but useful way to retrieve and
// handle any llvm::StringRef (or other llvm-internal strings)
// raw_string_ostream writes to the underlying string variable
llvm::raw_string_ostream rsoFeatName(featName);

// Load "isax.feature.name": !{!0}
// For a Function* F this would only be F->getMetadata("...")
if (auto md = MOD->getNamedMetadata("isax.feature.name"))
  // Load the first "element": !0 := !{!"DSP"}
  if (auto mdLevel1 = md->getOperand(0))
    // We know that this element, also has an "element": !"DSP"
    // Note: This could generally be a list
    // But here we can directly cast to llvm::MDString
    if (auto mdTyped = dyn_cast<MDString>(mdLevel1->getOperand(0)))
      rsoFeatName << mdTyped->getString();

// Make sure rsoFeatName writes possibly buffered data
// using a call to .str() or .flush()
rsoFeatName.flush();

// Done: featName will now hold 'DSP'
```

Listing 25: MetaData loading example of `ex_isax.ll`

ISAX metadata nodes

The following nodes are used in the generation of ISAX-wide information, like file or function names and e.g. the command line argument to request the feature in `clang`. By convention, place named metadata nodes at the top of an LLVM module and anonymous (”numbered”) definitions at the bottom (cf. Listing 24 lines 1-2 and 14-16).

- `!isax.feature.name`
  - The short name of the ISAX, which will be used to form the vast majority of generated identifiers
  - Expected: List with a single string element, e.g. `!{!"DSP"}`
  - Example: `-march=[...]x`**`DSP`**`0p1`
- `!isax.feature.prefix`
  - A feature prefix for the provided ISAX
  - `'x'` should be preferred, since it conveys specific meaning: *non-standard user-level extension*
  - Other options and their description can be found in ”getExtensionTypeDesc(StringRef Ext)” of: ”./clang/lib/Driver/ToolChains/Arch/RISCV.cpp”
  - Expected: List with a single string element, e.g. `!{!"x"}`
  - Example: `-march=[...]`**`x`**`DSP0p1`
- `!isax.feature.suffix`
  - An optional feature suffix for the provided ISAX (default: ””)
  - `'v'` (or empty) should be preferred, since it will be followed by the ISAX version
  - Expected: List with a single string element, e.g. `!{!"v"}`
  - Example: `-march=[...]xDSP`**`v`**`0p1`
- `!isax.feature.description`
  - A short description string of the provided ISAX (e.g. 1-2 sentences)
  - Expected: List with a single string element, e.g. `!{!"Some DSP description text."}`

- `!isax.feature.version.major`
  - An optional major version number (default: `"0"`)
  - This value has to be used each time when referring to the ISAX
  - Expected: List with a single string element, e.g. `!{!"0"}` (must be convertible to `uint`)
  - Example: `-march=[...]xDSP**0**p1`
- `!isax.feature.version.minor`
  - An optional minor version number (default: `"1"`)
  - This value has to be used each time when referring to the ISAX
  - Expected: List with a single string element, e.g. `!{!"1"}` (must be convertible to `uint`)
  - Example: `-march=[...]xDSP0p**1**`
- `!isax.asm.prefix`
  - An optional assembler prefix (default: (lowercase) `!isax.feature.name`)
  - When producing assembler output each instruction will be prefixed with this and a `"."`
  - Expected: List with a single string element, e.g. `!{!"dsp"}`
  - Example: **`dsp.`**`mac a2, a1, a0`
- `!isax.prefix`
  - An optional prefix (default: `"experimental-"`)
  - LLVM-internally `"x"` prefixed ISAXs will use the default (therefore, we recommend to use it as well)
  - Expected: List with a single string element, e.g. `!{!"experimental-"}`
  - Example: see e.g. output of `./llc -march=riscv32 -mattr=help` (LLVM with RISC-V support)

- **Special case:** (custom) registers
  - Append `!isRegister` to a global variable definition followed by an MDNode reference `!N`, where N is the corresponding identifier
  - Expected: List with two string elements
    `[0]` if ”standard”: the register is ignored w.r.t. its definition
    `[1]` the desired register kind name
  - Example:

```
@X = global [32 x i32] zeroinitializer, !isRegister !1
!1 = !{!"standard", !"GPR"}
!2 = !{!"non-standard", !"FPR42"}
```

Instruction metadata nodes

`"inst.def."` nodes will affect TableGen class generation, while `"intrinsic."` nodes allow us to provide signedness properties (when calling the actual builtin in C++) and override automatic deduction of specific information. Place these nodes between the function’s signature definition and body (cf. Listing 24 line 10).

- `!isax.func.inst.def.encoding`
  - The actual instruction bit pattern as space-separated string, using parameter names of the corresponding function signature (`rs1`) or binary literals (`3'b101`)
  - Generation of TableGen classes depend on this information (inluded by `./llvm/lib/Target/RISCV/RISCVInstrInfo.td`)
  - Possible collisions (e.g. caused by duplicate encodings) will be reported upon compilation of LLVM
  - Expected: List with a single string element (see example below)
  - Example: `[...] @lb(i5 %rd, i5 %rs1, i12 %imm)` could use `!{!"imm[11:0] rs1[4:0] 3'b000 rd[4:0] 7'b0001011"}`

- `!isax.func.inst.def.format`
  - Short format name used by the instruction
  - Possible values can be found here:
    `./llvm/lib/Target/RISCV/RISCVInstrFormats.td`
    See `class InstFormat` derived instances like `InstFormatR`
  - Note: Provided string will be appended to `InstFormat`
  - Expected: List with a single string element, e.g. `!{!"R"}`
- `!isax.func.inst.def.ins`
  - Input parameter types on instruction level (space-separated register or immediate types)
  - Possible values for registers (e.g. `GPR`, `FPR32`) can be found here:
    `./llvm/lib/Target/RISCV/RISCVRegisterInfo.td`
  - Possible values for immediates (e.g. `uimm5`, `simm12`) can be found here: `./llvm/lib/Target/RISCV/RISCVInstrInfo.td`
  - Expected: List with a single string element, e.g. `!{!"GPR simm12"}`
- `!isax.func.inst.def.outs`
  - Output parameter types on instruction level (space-separated register types)
  - Expected: List with a single string element, e.g. `!{!"GPR GPR"}`
- `!isax.func.inst.def.scheduleInfo`
  - Scheduling information (comma-separated), which helps to utilize functional units of the processor
  - Note: According to our information, the processor itself or the LLVM RISC-V backend still would have to implement e.g. *scoreboarding* [5] to avoid hazards (RISC-V processor example: [7])
  - Possible values (e.g. `WriteIALU`, `ReadFMul32`) can be found here:
    `./llvm/lib/Target/RISCV/RISCVSchedule.td`
  - Expected: List with a single string element
    e.g. `!{!"WriteIALU,ReadFMul32"}`
- `!isax.func.intrinsic.ID`
  - Optional name override for C++ intrinsics (default: function's name)
  - Expected: List with a single string element, e.g. `!{!"alt_mac_name"}`

- `!isax.func.intrinsic.paramTy`
  - Optional input type(s)[3] override (comma-separated) for C++ intrinsics
  - Note: Must be provided, if no function definition is available
  - Expected: List with a single string element
  - Example: `!{!"llvm_i32_ty,llvm_i32_ty"}`
- `!isax.func.intrinsic.properties`
  - Optional intrinsic function property[3] override (comma-separated) for C++ intrinsics
  - Note: Must be provided, if no function definition is available
  - Expected: List with a single string element, e.g. `!{!"IntrNoMem"}`
- `!isax.func.intrinsic.retTy`
  - Optional output type(s)[3] override (comma-separated) for C++ intrinsics
  - Note: Must be provided, if no function definition is available
  - Expected: List with a single string element
  - Example: `!{!"llvm_i32_ty,llvm_i32_ty"}` (usually single type)
- `!isax.func.intrinsic.signedness`
  - Signature information w.r.t. signedness for C++ intrinsics
  - Possible values: `"U"` (unsigned), `"S"` (signed), default: `"N"` (none / no information)
  - Order: Returned value's are followed by parameter's sign-info
  - Expected: List with a single string element
  - Example: `!{!"USSN"}`, (assume one return value) unsigned return, while the first two input parameters are signed and there is no information necessary regarding the third parameter

[3] Definitions (e.g. `llvm_i32_ty`, `IntrNoMem`) can be found here: `./llvm/include/llvm/IR/Intrinsics.td`

3.2. Automatic Deduction of Instruction Properties

With user-provided information via metadata only, we could already support an ISAX with conservative assumptions about the behavior of C++ intrinsics and their corresponding instructions. Either the ISAX has to convey intrinsic related information directly using metadata or implicitly by using e.g. registers in reading or writing fashion. The main idea is to mimic the instruction behavior inside the LLVM function definition. To deduct the actual parameter or return types automatically, we have to iterate each contained `Instruction`, query their parameters and check if they are in fact (e.g.) a register. While doing this we can also check certain memory properties and try to calculate higher level information about the used instruction. We will now look at the current subset of supported properties and then provide a small example of both use cases. One metadata-driven, the other fully automatic with special regard to intrinsic properties (Listing 26).

When defining intrinsics in the LLVM frontend or instructions at the backend we may provide high-level information on their respective behavior. In our work we chose the following properties, which can be checked for each `llvm::Instruction`[4] within the provided LLVM-IR representation: Memory accesses and exceptions. These can be queried by using `mayReadFromMemory`, `mayWriteToMemory` and `mayThrow` on the `Instruction` instance and allow us to compute further properties as depicted in Table 3.1. If for example every `llvm::Instruction` contained by our ISAX instruction, does not read from memory, we will assume that this holds true for the actual implementation. On the other hand: A single violation of this property will cause our tool to assume that there is in fact a memory access. Our default assumptions were chosen accordingly, to simplify the work with available getter-methods.

Table 3.1.: Set of gathered instruction information

Property	Default	Query / Computation
`MAY_READ_MEM`	`false`	`llvm::Instruction::mayReadFromMemory`
`MAY_WRITE_MEM`	`false`	`llvm::Instruction::mayWriteToMemory`
`MAY_THROW`	`false`	`llvm::Instruction::mayThrow`
`MAY_HAVE_SIDE_EFFECTS`	`false`	`(MAY_WRITE_MEM | MAY_THROW)`
`IS_PURE`	`true`	`!(MAY_HAVE_SIDE_EFFECTS)`
`IS_CONST`	`true`	`!(MAY_HAVE_SIDE_EFFECTS | MAY_READ_MEM)`

[4] See: `./llvm/include/llvm/IR/Instruction.h`

Once this information has been gathered, we may translate it into the corresponding properties. For C++ intrinsics these are shorthand notations of their attributes[5] e.g. *const* (`'c'`), *pure* (`'U'`) and *nothrow* (`'n'`); used in `BuiltinsRISCV.def`. LLVM intrinsics will use properties[6] like `IntrReadMem`, `IntrWriteMem` and `IntrNoMem` in `IntrinsicsRISCV.td`. Lastly, TableGen instruction definitions[7] offer e.g. the input of `mayLoad`, `mayStore` and `hasSideEffects`, which are then used in `RISCVInstrInfo.td`.

Furthermore, we need to bridge the semantic gap between the types used in the instruction's LLVM definition and its corresponding C++ intrinsic signature. On LLVM-IR level, when using General Purpose Registers (GPRs) the signature will use `"i5"`, while the C++ intrinsic should expect `"i32"`. Hence, we have to either provide the intrinsic's input and output types using metadata or infer them from a provided instruction definition. Since the former was already described in Section 3.1, we will now continue on how to determine the actual data type automatically.

Consider`@mac` in Listing 26, which is a multiply-accumulate instruction sample implementation. The idea is to identify certain instruction connections like: (1) loading the value from (2) a `GetElementPtrInst` (short: `GEP`) that is referring to (3) a global variable which in turn is (4) a register (similar for a store). Examples for such a load or store can be found on lines 12 and 19 (Listing 26), respectively.

A possible approach to identify such connected instructions is LLVM's pattern matching[8]. It provides a very compact way of matching certain instruction types and can be easily extended as shown in Listing 27. Matchers will also *bind* the actual instruction to a provided pointer (mismatch will result in a `nullptr`). In our example on line 11 we combine two matchers: `m_Load` and our own `m_GEP`. This will return a non-null match only if the provided instruction was a load with a `GEP` argument. We can then conveniently check if certain other properties are true and determine if the provided instruction was in fact a register load (a store is roughly equivalent and will therefore be omitted). Additionally, this also offers a way to retrieve the actual register and infer its data type, since the corresponding global variable is available and can be examined. Lastly, knowing that a certain instruction is in fact a register access will improve the accuracy of our property determination. Because `GEPs` on registers are not memory accesses and can therefore be ignored with regard to our instruction information (Table 3.1).

[5] See (lines 71+): `./clang/include/clang/Basic/Builtins.def`

[6] See (lines 20+): `./llvm/include/llvm/IR/Intrinsics.td`

[7] See `"class Instruction"` (lines 476+): `./llvm/include/llvm/Target/Target.td`

[8] See : `.llvm/include/llvm/IR/PatternMatch.h`

```
@X = global [32 x i32] zeroinitializer, !isRegister [...]

; (1) Automatic property deduction
;     Properties are implicitly provided by definition
define void @mac(i5 %rd, i5 %rs1, i5 %rs2)
; [...]
{
entry:
  %regIdx_X = getelementptr [32 x i32], [32 x i32]* @X, i32 0, i5 %rs1
  %x = load i32, i32* %regIdx_X
  %regIdx_Y = getelementptr [32 x i32], [32 x i32]* @X, i32 0, i5 %rs2
  %y = load i32, i32* %regIdx_Y
  %regIdx_Z = getelementptr [32 x i32], [32 x i32]* @X, i32 0, i5 %rd
  %z = load i32, i32* %regIdx_Z
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  store i32 %tmp2, i32* %regIdx_Z
  ret void
}

; (2) Metadata properties
;     Properties are explicitly provided by metadata
;     Function definition may be left empty and register be omitted
define void @mac_metadata(i5 %rd, i5 %rs1, i5 %rs2)
; [...]
!isax.func.intrinsic.paramTy !{!"llvm_i32_ty,llvm_i32_ty,llvm_i32_ty"}
!isax.func.intrinsic.properties !{!"IntrNoMem"}
!isax.func.intrinsic.retTy !{!"llvm_i32_ty"}
{
entry:
  ret void
}
```

Listing 26: Instruction property examples

With this set of tools we are able to load or generate identical intrinsic and instruction properties for both ISAX instructions from Listing 26. But we have to *write* actual C++ code and TableGen records as discussed in Chapter 2 in order to add ISAX support to LLVM. A possible solution to this task will be presented in the form of *replacements* in the upcoming section.

```
#include "llvm/IR/PatternMatch.h"
// Additional definitions for llvm::PatternMatch.
namespace llvm::PatternMatch {
  /// GEP / GetElementPointer instruction matcher.
  inline bind_ty<GetElementPtrInst> m_GEP(GetElementPtrInst*& G)
    return G;
}

bool isRegisterLoad(const llvm::Instruction& I) {
  llvm::GetElementPtrInst* GEP;
  if (match(&I, m_Load(m_GEP(GEP)))) {
    // Outline of what has to be done, additionally:
    //   Check if GEP->getOperand(0) is a
    //     llvm::GlobalVariable with metadata "isRegister"
    //   Check if GEP->getOperand(1) is constant zero
    //   Check if GEP->getOperand(2) is a llvm::Argument
    // If all three checks returned true, this is a register load
  }
}
```

Listing 27: Determination of Register Accesses

3.3. Output format

Now that we have collected all the necessary data from a provided ISAX, we want to modify LLVM's sources. Since we are already dependent of a built LLVM, we chose `clang-apply-replacements`, which offers code formatting features. If a suitable `.yaml` file is provided to this tool, it will execute file modifications according to its content. The only downside to this application is, that the target file has to exist, before patch application. While we could always write directly to source files during execution, this allows us to e.g. review the intended changes (helpful for debugging) or transfer the `.yaml` file to another system.

To ease handling of string templates we decided to use `{fmt}`[9] for the purpose of string formatting. It allows for a simplified creation of our patches and handling of format strings, a usage example can be found in Listing 28. All patch templates will use the `{fmt}` format, which means that each `{}`-encased string is in fact a string parameter.

[9] See: `https://github.com/fmtlib/fmt`

```
// {fmt} example: Define boolean variable 'HasNonStdExtDSP'
// Positional argument
std::string hasFeature1 =
  fmt::format("bool HasNonStdExt{} = false;", "DSP");

// Named argument
std::string fmtString_hasFeature =
               "bool HasNonStdExt{FeatName} = false;"
std::string hasFeature2 = fmt::format(fmtString_hasFeature,
                                      fmt::arg("FeatName", "DSP"),
```

Listing 28: `{fmt}` usage example

The following file format is imitating the one that is utilized when using `LibTooling` / `clang::tooling::Replacements`. Since we had no intention to use e.g. preview of changes or so-called `FixItHints` we decided to write the corresponding files directly, instead of pushing the data into `LibTooling` structures. In Listing 29 and Listing 30 (which is inserted into the former), we can see that seven parameters are used in total to describe a patch.

The most important ones being `TargetFilePath`, `ReplacementText`, `FileOffset` and `ReplacementLength`. Note that the remaining parameters are only used to store debug information e.g. of its intended purpose or generating method. `FileOffset` and `ReplacementLength` were deducted by using `LibTooling` in Section 2.5, while the `ReplacementText` has to be built e.g. by using templates, as shown in the usage example (Listing 28). There, we implicitly generated the patch content for `./clang/lib/Basic/Targets/RISCV.h`, if the metadata node `!isax.feature.name` was set to `"DSP"`. To modify multiple positions within the same file, we use Listing 30 for each position and concatenate each single replacement. Afterwards, the resulting string is provided as parameter `{Replacements}` into the format string described by Listing 29.

Once all patch files have been generated and put into a directory, a call to `clang-apply-replacements` (with said directory as argument) will execute the prepared modifications. If LLVM has been set up with `cmake` and `ninja`, we may now call the latter to re-build the target LLVM.

After this overview of our approach, we want to present the related work of `OpenASIP` and a manual implementation of ISAX support for LLVM.

```
---
MainSourceFile:  '{TargetFilePath}'
Diagnostics:
  - DiagnosticName:  {PatchName}
    DiagnosticMessage:
      Message:         '{PatchMessage}'
      FilePath:        '{TargetFilePath}'
      FileOffset:      0
      Replacements:
{Replacements}
    Notes:
      - Message:         ''
        FilePath:        ''
        FileOffset:      0
        Replacements:    []
    Level:           Warning
    BuildDirectory:  '{PatchFileDirectory}'
...
```

Listing 29: Replacement file template

```
        - FilePath:        '{TargetFilePath}'
          Offset:           {FileOffset}
          Length:           {ReplacementLength}
          ReplacementText: '{ReplacementText}'
```

Listing 30: Single replacement (template), which may be placed into {Replacements} of Listing 29

4. Related Work

4.1. OpenASIP / TCE tools

When looking at another architecture, the so-called Transport Triggered Architecture (TTA), we will now consider the OpenASIP[1] project. Its corresponding contribution is the TTA-Based Co-Design Environment (TCE) [1], which offers a wide spectrum of capabilities regarding the development of Application-Specific Instruction-Set Processors (ASIPs). A RISC-V processor that implements an application-specific or custom ISAX can be considered an ASIP.

Computations in a TTA are performed by Functional Units (FUs), the actual hardware implementation of an application specific instruction. Its execution can then be *triggered* by writing the input data onto a *triggering port*, similar to the data transport of a register to corresponding hardware units in RISC-V. However, the most distinguishing feature of TTAs is that FUs may directly communicate among each other, since it is possible to write an FU's computation result into another FU's triggering port. This in turn allows for flexible and modular processing of data. Additionally, it simplifies the hardware, since control flow may be predicted to a greater extent at compile-time.

TCE offers a complete design-environment, with conversions from high-level C++ program representations to actual hardware synthesis. Naturally, TCE allows the definition of custom operations, which may be designed and described using a standalone tool called `OSED`. These descriptions are stored in databases that hold information like we collected in Section 3.2 (e.g. memory accesses) and can be provided using C++ source code.

With regard to our work, actually utilizing these custom instructions is done in a very similar way. The developer has to call a C/C++ intrinsic to issue the corresponding operation. Furthermore, LLVM is used as the compiler framework and is being patched, but this is done to achieve compatibility between TCE and certain LLVM versions. However, in contrast to our work, compiling these sources requires not only LLVM but also TCE to be present.

[1] `http://openasip.org/`

4.2. Manual LLVM Integration of the Pulpissimo ISAX

Compiler support for any used ISAX is, like we discussed in the beginning of this work, a desirable feature. For completed ISAX, where (probably) no further changes will occur, this is usually done manually. A recent example in the RISC-V environment is related to the PULPissimo[2] architecture [4] / ISAX, implemented e.g. by the RI5CY core[3]. In [3] the implementation of a PULPissimo LLVM backend was presented.

PULPissimo offers a wide range of instructions, like the presented `mac` and corresponding variants, hardware loop support and vector calculations (among others). The implementation of its backend was correspondingly extensive and provided a starting point for our work. Since each instruction was known and domain-specific knowledge could be used, this backend offers support for each instruction and implements replacement patterns. Last mentioned patterns[4] allow matching specific code sequences and replaces them by an instruction, accordingly. An example would be to replace `add(mul(a,b),c)` by `mac(a,b,c)`.

Furthermore, this allowed the author to implement an instruction selection, which would automatically select suitable instructions for a given source code. However, they also faced a similar challenge with regard to C++ intrinsics. A subset of PULPissimo instructions was only supported when using GCC as compiler at that time. Thus, they added `clang` frontend support for C++ intrinsic function calls, which had to be matched to specific instructions in their case. Then a file prepared by GCC was provided to LLVM which would finally select the desired or compatible instruction.

While we are not able to cover the whole spectrum of PULPissimo, we will now proceed with a prudent comparison to their approach and general evaluation of our work.

[2] See also: `https://github.com/pulp-platform`

[3] `https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf`

[4] May be defined / included in `"./llvm/lib/Target/RISCV/RISCVInstrInfo.td"`

5. Evaluation

Our presented infrastructure ultimately generates patches for an LLVM project, which we will now examine, both quantitatively and qualitatively.

In order to support a given ISAX manually, developers have to modify (at the very least) five C++ source files and two TableGen files. While this might seem like a negligible requirement, recurring changes at ISAX level may introduce errors each time, since the sources have to be changed accordingly. Furthermore, malfunctions can be generally hard to detect and may only occur under certain circumstances. For example, if a C++ intrinsic's signature is not updated correctly, it might still remain executable and may only fail when certain types (compliant to the incorrect signature) are used. Since these modifications are scattered across the LLVM pipeline and have to be done in potentially large files, an automated approach is less error-prone. Another advantage is the reduced time required to adjust the ISAX support after each change, e.g. input files with 200+ instructions are processed in well under 20 seconds[1]. Most of the patch generation time is spent handling the C++ AST-matching, while the actual application via `clang-apply-replacements` takes up virtually no time. Naturally, a developer also has to put time in writing and maintaining corresponding input file(s). However, the presented ISAX description format using LLVM-IR metadata is malleable and may be extended or changed to ease writing or even allow automated generation from a hardware description.

A comparison of our approach with [3] proves to be not directly possible, since we support only `"R"`-type instructions and certain data types. Furthermore, the PULPissimo backend could use domain-specific knowledge to group instruction classes and exploit certain conditions to reduce implementation size. Nevertheless, we want to discuss the development effort conserved by our infrastructure in terms of Lines of Code (LoC). Therefore, we removed any comments and formatting newlines from our replacement strings to receive a lower limit. We found that for basic ISAX support (no new instructions) merely 21 LoC are required. Each added instruction will (e.g. dependent on argument count) require at least another 20 LoC. Experimental support for custom register fields could be achieved with only 6 LoC plus index count, to establish each individual register.

[1] sample executions were conducted using an AMD Ryzen 7 2700X CPU

Thus, depending on the scope of a provided ISAX, generated patches will easily introduce 1000+ LoC – most of them in TableGen files, more precisely: classes. However, this had no observable negative effect on LLVM's compilation time or behavior, despite the correspondingly high TableGen class count.

Since we introduced this infrastructure to allow for automated compiler support of RISC-V ISAXs, our approach seems to achieve reasonable results. When providing supported ISAX, our tool may speed up and simplify certain development cycles, especially when changes to the ISAX occur in fast succession. Additionally, it lowers the required domain-specific knowledge with regard to the actual LLVM source files, which is essential for modifying these files.

Finally, we will summarize this work and discuss possible extensions towards an improved automated ISAX support generation in the next chapter.

6. Conclusion

Application-specific processors are present in a plethora of everyday objects: from cars to mobile phones. Accordingly, the development of such processors is important and has a high impact with regard to many areas. The development of these architectures can greatly benefit from open-standard ISAs when combined with open-source compiler technologies, like RISC-V and LLVM, respectively.

Hence, we utilized the aforementioned projects and presented a possible way of implementing an infrastructure that may assist this development. Usually, custom ISAXs are not supported by a compiler and require manual implementation or e.g. inline-assembler calls to actually utilize its instructions. Therefore, we suggested implementing the compiler support automatically, using an ISAX description provided by a developer, without the need for manual LLVM source modifications.

By offering an automatic approach to perform precise source transformations within LLVM we are able to offload this fraction of ISAX development effort onto a tool. Implementing these functions requires solid knowledge of LLVM's pipeline stages on source file level. To achieve this goal, we suggested analyzing and modifying C++ files with tools provided by LLVM itself and presented how these can be used e.g. to identify a certain source code position.

If present, our approach will also analyze instruction definitions to derive the corresponding properties automatically. Combined with user-provided information regarding the actual ISAX we are able to assemble C++ and TableGen code within our tool. These pieces of code are then packaged into patches, which will implement the actual ISAX compiler support after they are applied. This will ultimately enable the developer to utilize the new instructions via C++ intrinsic function calls within a short amount of time while requiring less LLVM-specific knowledge.

Since our approach established an infrastructure to manage patch methods and manipulate C/C++ sources, there are many ideas for additional features. We will now present some small ideas and a major feature which would impact the actual RISC-V core support.

Outlook and Future Work

While testing different experimental features we have already implemented a basic support for custom registers, which has to be tested further and extended where necessary (see Section 2.2 and Section 2.4).

Furthermore, the supported instruction formats could be extended by adding the automatic definition of custom data types, like immediates with user-defined bit-width. One solution to this could be to ignore already available definitions, since we know the target LLVM's data types or may look them up. We can then place or include custom definitions in the corresponding file `"RISCVInstrInfo.td"`. Data type examples are e.g. the definition of `"uimm5"` and `"simm12"`.

Another major feature we explored was the handling of *hazards*, i.e. instruction pipeline problems which may lead to incorrect computation results. An example would be if our `mac` instruction took N clock cycles to actually compute and the processor would execute multiple calls to `mac` in less than N cycles. To check the actual behavior, we defined a custom processor in `"./llvm/lib/Target/RISCV/RISCV.td"`. Furthermore, a possible hazard was introduced if an instruction was to be defined with the scheduling info `WriteIALU` (see Listing 31).

When using multiples of such instructions in direct succession we could not observe any handling of the hazards in assembler output. Our expectation was that `NOPs` would be introduced by LLVM's scheduler, which was not the case. Other already existing scheduling models showed the same behavior and did not avoid hazards according to our information. Hence, we drew the conclusion that there is currently no general mechanism in the RISC-V LLVM backend to account for these situations.

Further investigation indicated that a corresponding scheduling algorithm exists e.g. for MIPS using delay slots, see `"./llvm/lib/Target/Mips/MipsDelaySlotFiller.cpp"`. Another possible approach might be to implement a custom scheduling pass, deriving from `MachineScheduler` (see `./llvm/lib/CodeGen/MachineScheduler.cpp`). This pass would have to transform (e.g. by adding `NOPs`) the machine program using so-called *instruction itineraries* which are offered by the scheduling model and may be used to check for hazards.

That way, hazards would not have to be handled by e.g. hardware scoreboarding [7] and simpler RISC-V cores could be supported for a given ISAX. This in turn would simplify the hardware itself and the development of these processors, which is an interesting advantage.

```
// Information on how to get started w.r.t. hazards and scheduling
//
// (1) Define a scheduling model
//     See e.g. "./llvm/lib/Target/RISCV/RISCVSchedule.td"
//         and  "./llvm/lib/Target/RISCV/RISCVSchedRocket.td"
def TestModel : SchedMachineModel {
  let MicroOpBufferSize = 0; // Architecture is in-order
  let IssueWidth = 1;        // 1 micro-op is dispatched per cycle.
  let UnsupportedFeatures = [HasStdExtV, ... ];
}
// (2) Define which resource (TestUnitALU) is claimed during execution
//     if an instruction has scheduling info including "WriteIALU"
//     And set for how long -- here 10 cycles
//     Note: Values in brackets will eventually account for hazards
let Latency = 10, ResourceCycles = [10] in {
  def : WriteRes<WriteIALU, [TestUnitALU]>;
}

// WriteIALU may be used for an instruction def.
//   Define / include instructions within
//     "./llvm/lib/Target/RISCV/RISCVInstrInfo.td"
//   and add scheduling info, e.g.
//     Sched<[WriteIALU, ReadIALU, ReadIALU]>
//   to incur hazard information

// (3) Definition of an actual processor
//     < name, scheduling model, list of supported features >
//
//     The CPU may be selected as target, e.g. in llc, using:
//     ./llc -march=riscv32 -mcpu=test-rv32 [...]
def : ProcessorModel<"test-rv32", TestModel, [FeatureDSP]>;
```

Listing 31: Hazard handling primer

A. Appendix

Table A.1.: RISC-V assembler mnemonics for `x` and `f` registers (cf. Table 25.1 from [6])

RISC-V assembler mnemonics for `x` and `f` registers			
Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	–
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	–
`x4`	`tp`	Thread pointer	–
`x5`	`t0`	Temporary/alternate link register	Caller
`x6-7`	`t1-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments/return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller
`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f10-11`	`fa0-1`	FP arguments/return values	Caller
`f12-17`	`fa2-7`	FP arguments	Caller
`f18-27`	`fs2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller

Table A.2.: `MetaData` "Cheat Sheet"

`MetaData` nodes used			
Identifier	**Description**	**Example Value**	**Optional**
Module specific `MetaData` – prefix: `"!isax."`			
`feature.name`	Short feature name	`"DSP"`	no
`feature.prefix`	Feature name prefix	`"x"`	no
`feature.suffix`	Feature name suffix	`"v"`	yes
`feature.description`	Short feature description	`"DSP / MAC instruction set"`	no
`feature.version.major`	Major version number	`"0"`	yes
`feature.version.minor`	Minor version number	`"1"`	yes
`asm.prefix`	Assembler prefix of all instructions	`"dsp"`	yes
`prefix`	LLVM-internal feature prefix	`"experimental-"`	yes
Instruction specific `MetaData` – prefix: `"!isax.func."`			
`inst.def.encoding`	Encoding string	`""`	no
`inst.def.format`	Short Instruction Format	`"R"`	no
`inst.def.ins`	Input types	`"imm12 GPR GPR"`	no
`inst.def.outs`	Output types	`"GPR"`	no
`inst.def.scheduleInfo`	Scheduling information	`"WriteIALU, ReadIALU, ReadIALU"`	no
`intrinsic.ID`	Name Override	`"alt_mac_name"`	yes
`intrinsic.paramTy`	Input Type Override	`"llvm_i16_ty,llvm_i32_ty,llvm_i32_ty"`	yes
`intrinsic.properties`	Property Override	`"IntrNoMem"`	yes
`intrinsic.retTy`	Return Type Override	`"llvm_i32_ty"`	yes
`intrinsic.signedness`	Signedness information	`"USN"`	no
Register specific `MetaData`			
Append `!isRegister` to its definition followed by an MDNode reference `!N` Where `N` is the corresponding MDNode number, e.g. `!3 = !{!"standard", !"GPR"}` Using `"standard"` in the first index, will suppress the definition of this register The second index holds the register kind name, here `"GPR"` Example: `"@X = global [32 x i32] zeroinitializer, !isRegister !1"` `"!1 = !{!"standard", !"GPR"}"` (standard registers will be *ignored*) `"!2 = !{!"non-standard", !"FP_REG"}"` (custom register)			

```
# Full example how the built LLVM could be used to compile a program
# source that is using C++ intrinsic functions and ultimately yields
# an assembler / object file

# Set paths (i.a. to make sure we use the correct LLVM)
LLVM_BIN_DIR="/path/to/built/llvm/bin"
CLANG="$LLVM_BIN_DIR/clang++"
OPT="$LLVM_BIN_DIR/opt"
LLC="$LLVM_BIN_DIR/llc"

# Target filename without '.cpp' extension
FN="dsp_test"

# The ISAX: <prefix><name><version>
# 'x' prefix is used for: "non-standard user-level extensions"
# Then the extension's name 'DSP' is followed by its version
# <MAJOR>p<MINOR> (here: 0.1)
ISAX="xDSP0p1"

# -- Build / concatenate clang parameters --
# Suppress two things: (1) optimization passes & (2) 'optnone' attribute
# Note: (2) would block any further (manually run) opt passes
PARAM="-O0 -Xclang -disable-O0-optnone"
# Our sample target is a 32bit RISC-V
PARAM="$PARAM --target=riscv32"
# Supply supported extensions / features of our target architecture
# Note: could add 'm' for integer-multiplication standard extension
PARAM="$PARAM -march=rv32i${ISAX}"
# Finally enable utilization of all extensions
# which are considered 'experimental' (e.g. 'x' or 'z' prefix)
PARAM="$PARAM -menable-experimental-extensions"

# Create LLVM-IR of our program (unoptimized, no register use)
$($CLANG $PARAM $FN.cpp -S -emit-llvm -o $FN.ll)
# Apply 'mem2reg' pass -> actually use registers (LLVM-IR)
$($OPT -S -mem2reg $FN.ll -o ${FN}_mem2reg.ll)
# Finally, create assembler (.s) and object / binary code (.o) files
$($LLC ${FN}_mem2reg.ll -o ${FN}_mem2reg.s)
$($LLC ${FN}_mem2reg.ll -filetype=obj -o ${FN}_mem2reg.o)
```

Listing 32: Sample `clang++` & `llc` call

```
# Setup of 'cmake'
# Taken from:
# https://clang.llvm.org/docs/LibASTMatchersTutorial.html
# See: "Step 0: Obtaining Clang"
# ----------------------------
cd ~/temp
git clone git://cmake.org/stage/cmake.git
cd cmake
git checkout next
./bootstrap
make
sudo make install
```

Listing 33: Possible way to obtain `cmake`

```
# Setup of 'ninja'
# Taken from:
# https://clang.llvm.org/docs/LibASTMatchersTutorial.html
# See: "Step 0: Obtaining Clang"
# ----------------------------
cd ~/temp
git clone https://github.com/martine/ninja.git
cd ninja
git checkout release
./bootstrap.py
sudo cp ninja /usr/bin/
```

Listing 34: Possible way to obtain `ninja`

```
# Recommended options:
# ----------------------------
cmake -G Ninja \
-DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" \
-DCMAKE_BUILD_TYPE=RELEASE \
-DLLVM_TARGETS_TO_BUILD="X86;RISCV" \
-DBUILD_SHARED_LIBS=ON \
$path_to_llvm_sources/llvm

# Additional, useful options:
# ----------------------------
# When using DEBUG build type
  -DLLVM_OPTIMIZED_TABLEGEN=ON
# ----------------------------
# Adapt resources (affects build time)
  # N := available CPU cores
  -DLLVM_PARALLEL_COMPILE_JOBS=N
  # Higher values cause significant increase in memory usage
  -DLLVM_PARALLEL_LINK_JOBS=2
# ----------------------------
# Select alternate compilers (e.g. clang-10)
  -DCMAKE_C_COMPILER=/path/to/clang-10
  -DCMAKE_CXX_COMPILER=/path/to/clang++-10
```

Listing 35: Recommended LLVM build options

```
// Examples of class arguments taken from "class Shift_ri"
//
// outs       -- Example: (outs GPR:$rd)
// ins        -- Example: (ins GPR:$rs1, uimmlog2xlen:$shamt)
// opcodestr  -- Example: "srai"
// argstr     -- Example: "$rd, $rs1, $shamt"
// pattern    -- Example: [] // mostly kept empty
// InstFormat -- Example: InstFormatI
//
// Note: in RVInst ''opcodestr'' and ''argstr'' are used as follows:
// let AsmString = opcodestr # "\t" # argstr;
//
// InstFormat may be:
// Pseudo, R, R4, I, S, B, U, J, CR, CI, CSS, CIW, CL, CS, CA, CB, CJ, Other

// Superclass used for definition of all ISAX instruction classes
class RVDSPISAXInst<dag outs, dag ins, string opcodestr,
                    string argstr, InstFormat format>
    : Instruction {
  field bits<32> Inst;
  // SoftFail is a field the disassembler can use to provide a way for
  // instructions to not match without killing the whole decode process.
  // It is mainly used for ARM, but Tablegen expects this field to exist
  // or it fails to build the decode table.
  field bits<32> SoftFail = 0;
  let Size = 4;

  let Namespace = "RISCV";

  dag OutOperandList = outs;
  dag InOperandList = ins;
  let AsmString = opcodestr # "\t" # argstr;
  let Pattern = [];
  let TSFlags{4-0} = format.Value;
}
```

Listing 36: Custom TableGen Instruction Class (see Section 2.4)

List of Figures

List of Tables

List of Acronyms

ABI	Application Binary Interface
AES	Advanced Encryption Standard
ARM	Advanced RISC Machine
ASIP	Application-Specific Instruction-Set Processor
AST	Abstract Syntax Tree
AVX	Advanced Vector eXtensions
CPU	Central Processing Unit
FU	Functional Unit
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
GPR	General Purpose Register
GPU	Graphics Processing Unit
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISAX	ISA eXtension
LoC	Lines of Code
LLVM	Low Level Virtual Machine
LLVM-IR	LLVM Intermediate Representation
MIPS	Microprocessor without Interlocked Pipelined Stage
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD eXtensions
TCE	TTA-Based Co-Design Environment
TTA	Transport Triggered Architecture

Bibliography

[1] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg. "HW/SW co-design toolset for customization of exposed datapath processors". In: *Computing platforms for software-defined radio*. Springer, 2017, pp. 147–164.

[2] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[3] S. Muhcu. *Implementierung eines Pulpissimo-Backends für die LLVM Compiler-Infrastruktur*. B.S. thesis. 2019.

[4] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini. "Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX". In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3.

[5] J. E. Thornton. "Parallel operation in the control data 6600". In: *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*. 1964, pp. 33–40.

[6] A. Waterman and K. Asanović. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213". In: (2019).

[7] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640.