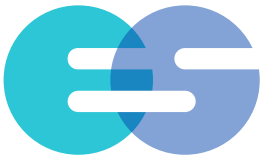

Hamburg University of Technology
Institute of Embedded Systems
Prof. H. Falk

Design and Test of a Code Selector Ruleset for Pointer Expressions targeting RISC-V Processors as part of a WCET-aware Compiler

Bachelor Thesis

Ruben Kuhlmann

December 12, 2022

Project Description

Hamburg University of Technology
Institute of Embedded Systems
Prof. H. Falk

Design and Test of a Code Selector Ruleset for Pointer Expressions targeting RISC-V Processors as part of a WCET-aware Compiler

The task for this bachelor thesis is to develop and extend the WCC upon a pointer ruleset for the RV32IMC implementation of the RISC-V instruction set architecture. After this work, the WCC is supposed to be able to translate C source code containing pointer expressions into assembly code executable on the aforementioned RISC-V architecture. For this, an already existing ruleset will be expanded with additional rules. Furthermore, auxiliary classes and functions will be added to aid these rules with the creation of assembly code. To ensure proper functionality everything implemented has to be tested and validated with an existing test framework.

In this thesis, first, the title and its keywords will be introduced, subsequently the implemented ruleset will be explained and lastly, the state and functionality of the pointer ruleset will be examined.

First Examiner	: Prof. Dr. Heiko Falk
Second Examiner	: Thilo Fischer
Advisor	: Prof. Dr. Heiko Falk
Due Date	: 11.12.2022

Declaration

This project is the result of my own work, except where explicit reference is made to the work of others, and has not been submitted for another qualification to this or any other university.

(Ruben Kuhlmann)
Hamburg, December 12, 2022

Abbreviations

ACET	Average Case Execution Time
WCET	Worst Case Execution Time
WCC	WCET-aware C Compiler
IR	Intermediate Representation
LLIR	Low-Level Intermediate Representation
WIR	WCC Intermediate Representation
ISA	Instruction-Set Architecture
RISC	Reduced Instruction Set Computing
opcode	Operation Code
ANSI-C	American National Standards Institute C standard

Contents

Chapter 1

Introduction

The title of this thesis reads: "Design and Test of a Code Selector Ruleset for Pointer Expressions targeting RISC-V Processors as part of a WCET-aware Compiler". The goal of this chapter will be to take a look at the fundamentals needed to understand and explain the title.
At the heart of this thesis stands the notion of "embedded systems" which is important to put this work into perspective and understand its value. Thus we will take a look at the notion in section 1.1. Afterwards, the motivation for this thesis and related work will be introduced in 1.2. In order to later evaluate accomplishments, the goals for the thesis are outlined in section 1.3 which will also go over work already done by others. And to finish the introduction in section 1.4, the structure of this thesis is described

1.1 Embedded Systems

The WCET (worst case execution time) -aware compiler and RISC-V make up the foundation of the thesis. For both the notion of ”embedded systems” is essential. As such this section aims to give a brief introduction into the topic and explain why it is so important as well as which implications this has. Unless otherwise noted, all information in this chapter has been taken from [1].
I would like to start this introduction of with a quote from Peter Marwedel. He describes the term ”embedded systems” as such: ”Embedded systems are information processing systems embedded into enclosing products” [1]. Here, examples for enclosing products reach from things like cars or plains into the medical field or even communication devices like cellphones. Another quote from [1], which further puts this topic into perspectiv: ”Information technology (IT) is on the verge of another revolution. [...] networked systems

of embedded computers [...] have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways. [...] The use [...] throughout society could well dwarf previous milestones in the information revolution." This quote illustrates the integration of information processing devices into an enormous amount of daily appliances like fridges, toasters, dishwashers etc. The quote also demonstrates how much impact embedded systems have and how large-scale their application is. This all goes to show how important embedded systems have become to our daily lives and to our society in general, insinuating a focus of development on this topic. All these different applications share a strong connection between the information processing device and the physical world. These kinds of devices are also commonly called a Cyber-physical system (CPS). As such, they have to be dependable, which means that the following attributes have to be met according to [1]:

- Security: system is confidential and provides information integrity
- Safety: no risk to the health of people
- Reliability: the system does not fail within longer runtime
- Repairability: if the system failed, it can be restored in a reasonable time period
- Availability: is the probability that the system is available during runtime

Most of theses attributes rely on the system meeting time constraints for computation. Peter Marwedel furthermore states that many cyber-physical systems are bound by real-time constraints [1]. Not completing computational tasks in certain time frames almost certainly leads to a loss of quality in the end product, examples for this could be audio or video applications. However in some cases, due to the nature of CPSes, physical harm may be caused to the human user, examples for this might be applications in things like cars, planes or in the medical field. One can easily see that because of these reasons, real-time constraints are immensely vital to the CPS / embedded system. These time constraints can be split into two categories: the first ones are called hard time constraints. This term is used if violating a time constraint possibly would lead to catastrophic events [1, p. 10]. Everything else is called a soft time constraint.

This leads to the importance of the WCET (worst case execution time). Nowadays, the vast amount of embedded systems have been paired with techniques to speed up average case execution time (ACET). This is represented

by the use of components like caches or branch prediction in processors. However, the influence this kind of ACET oriented applications have on the WCET are often unpredictable and might even lead to worse WCET. This has the consequence that developers often do not use optimization at all in their applications for embedded systems. Another quote from [1, p. 11] stresses this point: "Many modeling techniques in computer science do not model real time. Frequently, time is modeled without any physical units attached to it, which means that no distinction is made between picoseconds and centuries." This absence of WCET consideration is the motivation for the WCET-aware C Compiler (WCC) which shall be extended by the ability to compile C source code for the RISC-V architecture, which in this case can be understood to be the information processing device, i.e., the embedded system at hand.

1.2 Related Work and Motivation

As compilers build the bridge between the human readable description of a program and the generated machine code, they play an essential role in software design and have the potential for substantial amounts of WCET optimizations and tracking. However as is noted in [1], many modern compilers and information processing systems in general are only concerned with improving the average execution time, which often leads to unknown or even negative effects for the WCET. This endangers real-time systems and their computational tasks which in turn shows the necessity for the WCET-aware compiler. As disregarding the WCET may have catastrophic consequences. However a good oversight over the WCET may even reduce product costs as it is easier to gauge the computational power needed for the application in which the embedded system will find itself in.

As of now the WCC supports the TriCore processors of the TC179x family and the ARM7 processors as target architectures. The TriCore processors are used in the automotive field, while ARM processors are for example universally used in smartphones. Currently, support for a third architecture, namely RISC-V is being worked on. RISC-V manages do differentiate itself from many other instruction set architectures (ISA). Among other things by being open source and adaptable, therefore it already finds many applications in the industry. In chapter 2.4, the architecture will be further introduced.

The work on the RISC-V implementation for the WCC was started by Jonas Oltmanns during the year 2021 [2]. He expanded the low level intermediate representation WIR to include the RISC-V instruction set architecture. The WIR was expanded upon the necessary assembly instructions and as-

sociated opcodes. Furthermore, the registers as well as immediates used by RISC-V were implemented in the WIR. Maurice Hoffmann continued in 2022 with his thesis [3]. During his work he implemented and evaluated a true bit value flow analysis for RISC-V. This included the implementation of a top-down and bottom-up analysis of the instructions from the RISC-V instruction extensions RV32I and C. These extensions will be explained in 2.4. In the project "RISCY" [4], conducted in 2022, the basic infrastructure for the code selector was set up. This included the introduction of an instruction factory and the code selector itself. Furthermore the implementation of a basic ruleset able to compile integer arithmetic and the modification of so called make files and similar, to make calling the RISC-V implementation possible. Furthermore a register allocator was constructed. After this, in 2022, Christian Sühl [5] set up a functioning stack together with the necessary framework. This made the RISC-V implementation able to successfully compile function calls and passed arguments, as well as returning properly from functions. There was clearly a lot of work done already, however many more things like pointer, composed data types, type extensions and much more still need to be implemented for RISC-V.

1.3 Goals for this Thesis

This thesis aims to tackle the support for pointers. The specific task is to extend the compiler by the ability to translate C pointer statements to assembly code fit to be executed on a RISC-V processor. The goal is to create a ruleset (further explained in section 2.3) which extends the compilers already existing ruleset to successfully compile pointer operators such as the address operator & and the indirection operator *. Furthermore, pointer-arithmetics shall be implemented such that things like ($+$, $-$, $+=$, $-=$, $++$, $--$) are also known to the WCC. Lastly, pointer as function arguments and returning functions are to be operational as well. In addition to this, as many pointer-tests of the builtin test-bench as possible should be able to compile successfully. The test-bench will be further examined in section 4.2.

1.4 Outline of this Thesis

Chapter 2 "Technical fundamentals" aims to further introduce the topic at hand. For this purpose, the technical side is looked at, meaning we will first see in section 2.1 what an compiler actually is and then introduce the WCC and how it is structured in sec. 2.2. In the following, this introduction

is then used to see where specifically this work lives in the code and what already was present in terms of functionality for the RISC-V architecture. To finish the previous chapter, ICD-CG is introduced in sec. 2.3. Here, the fundamentals for actually implementing the ruleset and what this term means will be explained. After this, the target architecture RISC-V shall be explained in sec. 2.4. Here, a briev introduction into hardware architectures in general is given. Finally, this leads into an introduction to the instruction extensions RV32I, M and C. To introduce all terms mentioned in the thesis title, pointers in the C language are explained in section 2.5.

Chapter 3 goes into the creation of the contribution this thesis makes to the WCC. For this, later used classes and functions are explained in sections 3.2 and 3.3. In the following, the outlines of the implemented rules are introduced in sections 3.4 and 3.5. Finally, the rules implemented are introduced and explained in sec. 3.6.

Chapter 4 "Experimental Results" serves for quantification of the accomplished work and includes testing for correctness. The first section 4.1 in this chapter does so by the use of manual tests. The second chapter 4.2 uses the testbench to test the ruleset implemented on a large scale.

Finally,chapter 5 "Conclusion and Outlook" denotes the end of this thesis and thus draws a conclusion of the work done as well as suggestions at possible future work to be done.

Chapter 2

Technical Fundamentals

In this chapter, first compilers in general are introduced in sec. 2.1. Then, building upon that, the WCC is introduced in sec. 2.2. Section 2.3 aims to complete the WCC part of this introduction, by explaining the framework for creating the rules. To introduce RISC-V, section 2.4 explaines the architecture and goes over the used instruction set extensions. Finishing this chapter up, section 2.5 takes a look at pointers in ANSI-C.

2.1 Compiler

As this thesis aims to expand a compiler, it is sensible to take a look at the meaning behind this term. In "Compiler Aufbau und Arbeitsweise" [6], it is stated, that in order to interface with a computational device, it is important to unambiguously express oneself in the creation of computational tasks. To do this while maintaining a human readable terminology is a challenge, solved by programming languages. Usually, the closer these languages get to the terminology of a specific application, the further they stray from maintaining execute ability by the computational device. In before a program is executable, its application specific description needs to be translated back into machine specific code, to make the program executable. This is the task of a compiler.

2.2 WCC

The WCET-Aware C Compiler (WCC) forms the foundation of this thesis. It is a compiler framework which compiles C sources to various instruction set architectures. As indicated by its name, the WCC aims to enable an effective and automatic WCET minimization with the means of a static WCET

analyzer [7] called aiT [8]. Currently supported target architectures are the TriCore processors of the TC179x family, the ARM7 processors and RISC-V which shall be expanded during this work. As already noted, the TC179x and ARM7 already have a complete implementation, while RISC-V support is still in work.

To introduce the WCC, first the general structure is outlined in sec. 2.2.1, after which a closer look is taken at the intermediate representations used in subsection 2.2.2, as these play an integral role in the construction of the ruleset. With an idea of the inner workings of the WCC, subsection 2.2.3 shows the current state of the RISC-V implementation. To end the WCC chapter, subsection 2.2.4 serves to take a look at the implementation of the pointer ruleset of the other two architectures supported by the WCC.

2.2.1 Structure

In this section, the structure of the WCC shall be outlined, with deeper explanation of parts that are important to the implementation of the pointer ruleset. Unless otherwise noted, all of the information in this section has been taken from [7, p. 26]. The overall structure of the WCC can be seen in figure 2.1.

Compilers in general are commonly divided into a front end as well as a back end. This is similary done for the WCC, as can be seen in figure 2.1, where the bold arrows represent the workflow for a typical compiler. On this path of bold arrows, the front end consist of the first three nodes and the other four nodes belong to the back end. The path for an ANSI-C source code, that shall be compiled, is to first go through the front end, which consists of the ICD-C Parser present in the figure. This front end generates the High-Level ICD-C intermediate representation (IR) of the code. Then starting in the back end, this high level IR gets translated into a low level IR. The WCC has two possible IRs for this, either the Low-Level Intermediate Representation (LLIR), or the WCC Intermediate Representation (WIR). This low level IR is then run through a code generation phase, which generates the assembly code specific to the chosen hardware architecture.

2.2.1.1 Front End

The front end of the WCC consists of a parser which generates an intermediate representation of the code, to which ACET source code optimizations are applied [7, p. 26]. As a first step, the C source is run through the parser. Here, first a lexical, then a syntax and lastly a semantically analysis of the C code is performed. In the lexical analysis, the source code is scanned for

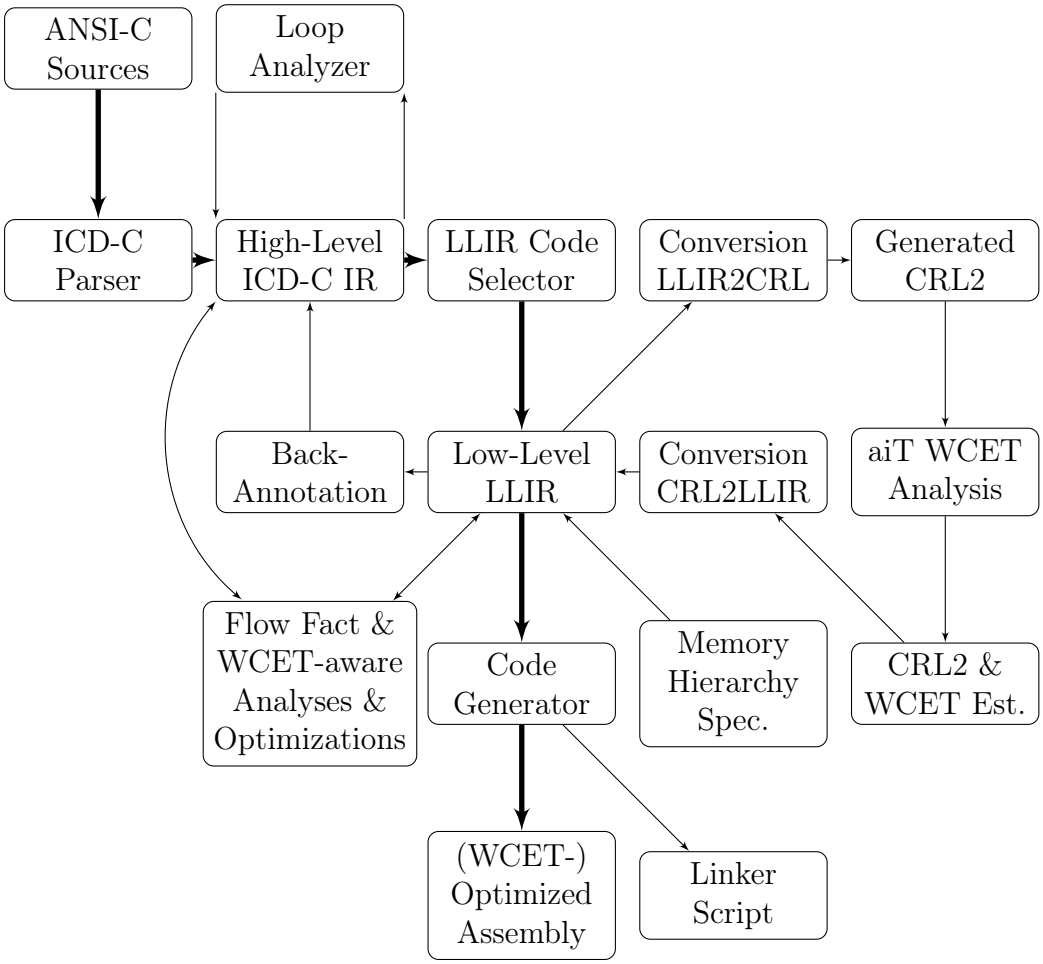

Figure 2.1: Workflow of WCET-aware C compiler WCC [7, p. 27]

and broken up into so called "lexemes", these are the strings of characters of significance for the source language, like "if" or "while" statements. Furthermore, keywords and constants are identified. These lexemes are then stored as tokens. During the syntactical analysis, the tokens are grouped hierarchically, which generates a structure of tokens resembling a tree. This structure is called a syntax tree. The buildup of this syntax tree is based upon the specifications of the C language [9].

In the semantic analysis this constructed tree is scanned for improper structures. Within this phase, the high-level IR (ICD-C IR) is finalized. ICD-C is still relatively close to the C source even allowing for transformation back into the source code [7, p. 27]. This whole front end is provided by the ICD-C compiler front end [10].

2.2.1.2 Back End

The back end phase begins with the given ICD-C IR. The first objective is to lower the abstraction level of the code. For this purpose, the ICD-C IR has to be translated into a low level IR. To accomplish that, assembly instructions have to be found. These instructions have to fully represent the replaced code constructs from ICD-C while minimizing costs. These costs are associated with each instruction. This translation is carried out with a code selector [7, p. 29]. The code selector is where most of the work associated with this thesis resides. It is based upon a tree-pattern matching approach, where a tree pattern is translated with a defined set of rules. The tree grammar has associated costs. With these costs the matcher generates an optimal parse of trees, with the most optimal and appropriate rules for each token in the tree to be translated [7] [5]. This translation results in virtual assembly code, in the form of a low level IR. The virtual assembly code is based on the idea that there are infinitely many registers to be used to increase flexibility [7, p. 31].

There are currently two possible outputs of the code selector to be generated. One is the Low-Level Intermediate Representation (LLIR) and the other is the WCC Intermediate Representation (WIR). While the TriCore architecture supports both outputs and the ARM architecture implementation only supports the LLIR output, RISC-V will only make use of the WIR, as it is the newer one of the two IRs.

With the newly generated IR, it is possible to perform different analyses, like control flow analysis, analysis of basic block relationships for things like reachability (explained in sec. 2.2.2.1). Further analyses include data flow consisting of live time and def/use chains [7, p. 30]. These analyses allow for processor-specific optimization and for further optimization goals besides the WCET, like ACET or energy consumption. Among those optimizations is register allocation which assigns the virtual register to physical registers. Since the RISC-V processor has a limited amount of registers, the virtual registers have to be mapped to the physical ones. The register allocation for RISC-V uses the idea of graph coloring [7, p. 32]. After these optimizations are applied, the code generator emits valid assembly code for the RISC-V RV32IMC implementation. This step is the final one for the compiler and thus the end of the back-end.

2.2.2 Intermediate Representations

Intermediate representations are commonly found in compilers and they are the internal data structures used to model/represent the source code. Their

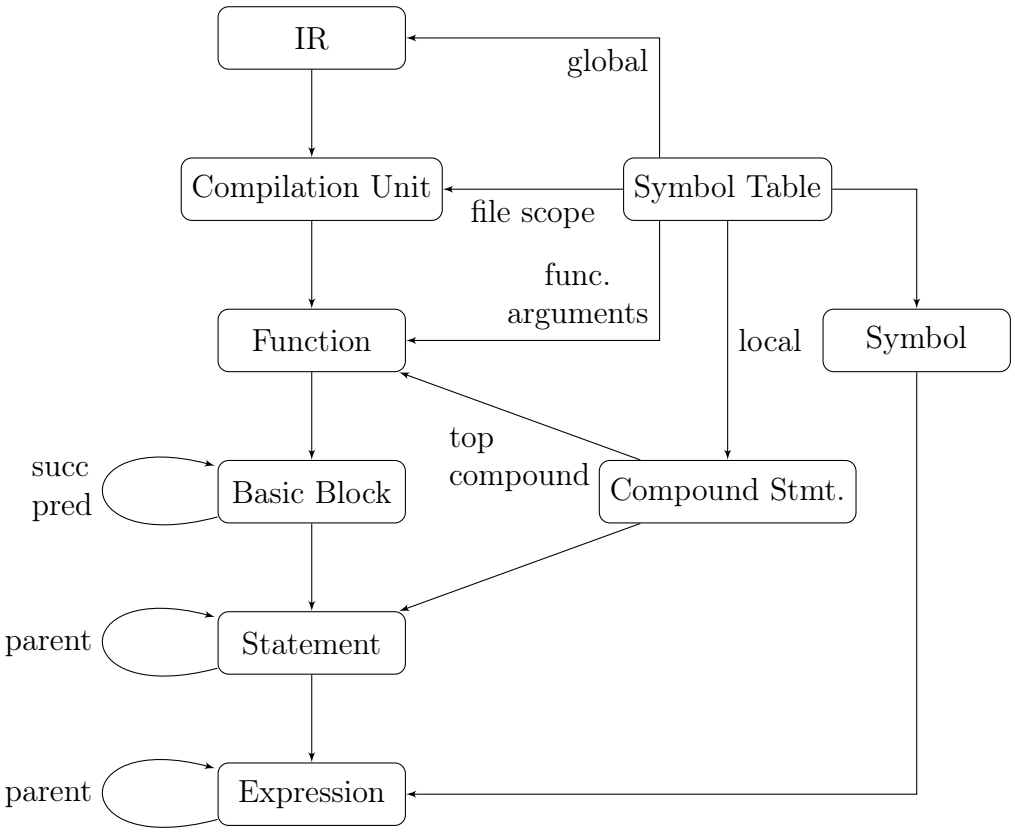

Figure 2.2: simplified class model of the ICD-C IR [7, p. 28]

use includes code analyses as well as code optimization [11]. Within compilers, there often are multiple levels of IRs used, differing in their closeness to the original source or the assembly code. In the following two subsections, first the high level IR ICD-C shall be introduced in sec. 2.2.2.1. Then, the low level IR WIR is looked at in sec. 2.2.2.2. These two IRs are represented here, as this thesis deals with both of them with the goal to be able to translate pointer expressions present in ICD-C into assembly instructions present in the WIR.

2.2.2.1 ICD-C

The ICD-C IR, as explained earlier, is used to represent C source code. Information in this section has been taken from [10]. A simplified class model of ICD-C can be seen in figure 2.2. Going down the route on the left side of the figure, the first class except the IR itself is the compilation unit. If a program to be compiled consist of different files, these files each are

represented by a compilation unit. The structures inside of these are then modeled with their respective functions, statements and expression classes of the IR, as displayed in the figure 2.2. Additionally, the IR implements a basic block class which is realized between the functions and the statements. A basic block represents a set of statements which do not divert the control flow, i.e., the set of statements in a basic block have exactly a single entry and a single exit point.

In addition, ICD-C uses symbol tables to save scope and data types of variables. As storage for arbitrary information, so called "persistent objects" are used. Since these can be attached to any IR construct, WCC uses them for example, to save WCET information for basic blocks [7, p. 28].

With this structure, ICD-C is able to perform numerous analyses and optimizations. Including data flow optimizations, loop transformations and interprocedural optimizations. The optimizations can be included or excluded when executing the WCC [7, p. 29].

To have an idea of what an ICD-C representation might look like for a given C code, a small example is introduced at this point. The example consist of the C code lines:

```
int i = 10;
int *p = &i;
```

Where first a integer variable i is instantiated with an value of 10. Then a pointer p to an integer variable is created and the address of i is saved to the pointer. These two lines of C code produce the ICD-C trees seen in figure 2.3.

The left tree represents the first line. The variable i is represented through the left leaf, i.e., through a symbol expression. The number 10 is represented through the integer constant expression on the right leaf. To represent the assignment of the number to the variable both leafs have an assignment expression parent. The second line is similarly represented with the right tree. Additionally an extra child node on the left side of the assignment expression is used to represent the address operator which is applied to i.

2.2.2.2 WIR

The WCC Intermediate Representation (WIR) is the low level IR used in the RISC-V implementation. Instead of resembling the C source code like the ICD-C, WIR represents virtual assembly code. This means, it already uses processor-specific instructions, however virtual register are used. Which in turn implies that, at this point in time, an infinite amount of registers is assumed.

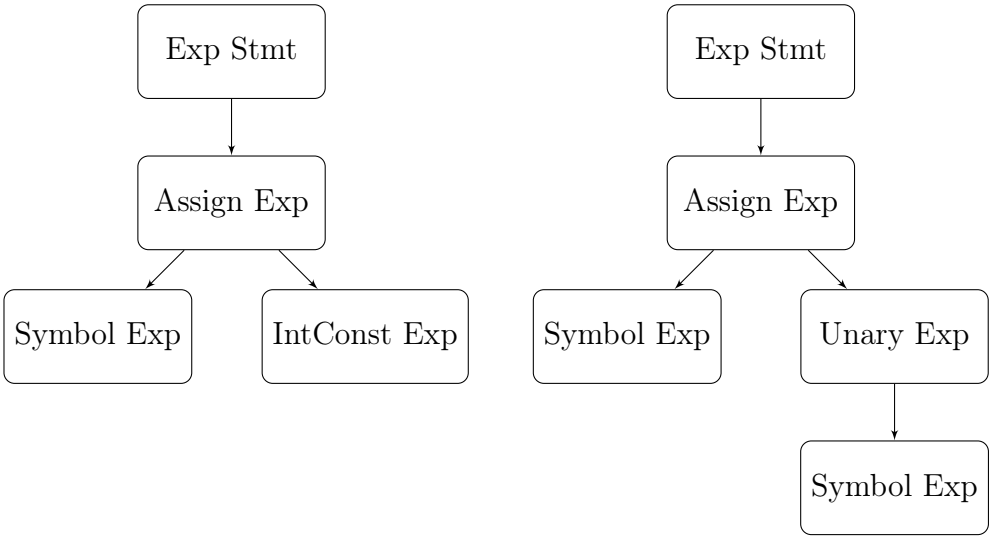

Figure 2.3: ICD-C tree example

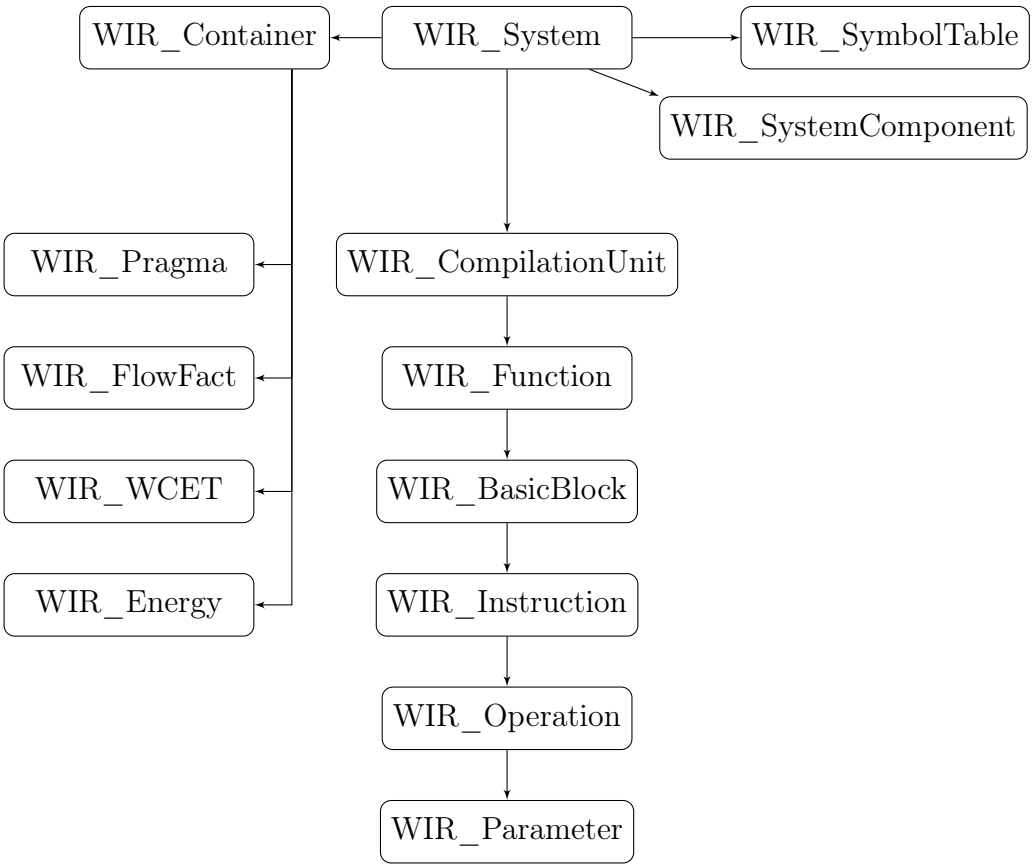

Figure 2.4: Simplified class model of WIR [4, p. 5]

In figure 2.4, a simplified structure of the WIR can be seen. For this thesis, the important part of the structure, is the path down the middle. The first component along its path is the WIR_CompilationUnit. This represents an entire assembly file. Further down, the WIR_Function represents assembly functions. The WIR_BasicBlock, like its ICD-C counterpart, represents a sequence of instructions which has exactly one entry and one exit point. The WIR_Instruction symbolises a set of operations which can be executed in parallel. Since RISC-V only executes one instruction at a time, this class does not matter much for this implementation. WIR_Operation now corresponds to exactly one machine operation, like ADDI. Finally, WIR_Parameter represents the parameters of an operation, like the used register, immediates or labels. [4]

2.2.3 Current State of RISC-V Support

As mentioned before, work is still very much ongoing for the RISC-V support by the WCC. The mentioned RISCY project [4] among other things, introduced the so called instruction factory. The instruction factory is important for this thesis, as it interfaces with the WIR. In case, a rule matching a certain part of a tree needs to generate some instructions or a new register, which need to be inserted into the WIR structure, the instruction factory is called. A lot of helper functions were created in the rv32_prolog.m4 file. These are often used in cost parts of the implemented pointer rules.

Furthermore, the RV32_AddressModification class was implemented, which pointer rules strongly rely on. The class is used to represent a register, holding an address, together with a pending modification for this address.

Another important class that was given, is the RV32_LValue class. It is used to represent lvalues from the C source code. For this they not only need to keep track of the lvalue's register, but also the associated memory location.

Furthermore, a ruleset was introduced. The rules present at the start of this work included general rules, dealing with integer arithmetic, assignments and relations. Many of these rules are being used in the tests for this work, as they deal with common C code expressions almost always used in any C program.

Christian Sühl, as mentioned, implemented the stack in [5]. His work was also crucial to this thesis, as the stack is called in some pointer rules and also necessary to call/return from functions. Functions are also frequently used in tests.

In general, the RISC-V implementation was already able to successfully compile C code dealing with integer type symbols at the start of this thesis.

2.2.4 Existing Implementations

As mentioned above, the other two architectures already have full support and thus they both have completed rulesets and auxiliary functions to handle C pointers. In the bachelor thesis of Tobias Marschner [12], this part of the WCC was implemented for the ARM architecture. Both implementations of the pointer ruleset rely on an addresmodification class. Furthermore, both distinguish between normal datatype registers and address registers. In the case of the TriCore, this has to be done, as here the data registers and address registers actually differ from one another. However in the ARM implementation, this is done purely for convenience. Furthermore, both architectures make use of an address with offset class.

2.3 ICD-CG

ICD-CG is responsible for translating the ICD-IR into the WIR, therefore it is at the heart of this work, which is why ICD-CG is introduced here. The information of this chapter has been taken from [12, p. 23] which is a good summary of [13, p. 129]. ICD-CG is a so-called code generator generator. This means that it is able to take a set of rules as input, using which the actual code generator is produced. The generated code generator is then able to translate the ICD-IR to the WIR. To perform said translation, ICD-CG makes use of tree pattern matching.

Tree pattern matching replaces a certain amount of terminals and nonterminals with a single new nonterminal *via* a specified set of rules for such transformations. The tree to be matched is the previously mentioned ICD-C syntax tree. Every node of this tree is made up of a terminal. The goal of the tree pattern matcher is to apply rules until the whole tree is reduced into a single nonterminal. If it is not possible to reduce the tree this way with the specified rules, the code generation fails.

Each rule in the ruleset consists of two parts [14], the first is an action part which constitutes the instructions to be added, as well as it calls the action parts of the terminals/nonterminals it matched against. Thus, invoking the action part of the single last nonterminal will invoke the action parts of all the other matched terminal/nonterminals, generating the WIR. The second part is a cost part, as there often are multiple possible rules that are able to match a specific sub-tree. These costs are made up of the costs of the rule's sub-tree, as well as the added costs from performing the rule's action part. This effectively makes a rule's cost equate to the entire cost of a sub-tree. Thus, the tree pattern matcher is able to decide which rule to use by

optimizing for an optimal cost. The structure of a rule looks as follows:

```
nonterminal : nonterminal | terminal | terminal (  )
{ cost } = { action };
```

The first nonterminal is the one created by applying the rule at hand, afterwards comes the tree structure that is to be matched. For this, there are three different possibilities. The first one is a transformation from one nonterminal to another one. The second one is the transformation from a single terminal, still not reducing the overall nodes of the tree. The last one matches against a terminal, which in turn has one ore more child nodes attached. The child nodes are separated by a comma: terminal (nonterminal, nonterminal, ...) [14].

With respect to the cost and action parts, one can use OLIVE-specific identifiers. This is due to ICD-C being OLIVE compatible. These start with a $ and allow the C++ code used within these parts to interface the pattern matcher. Important are:

- $N which refers to the nth node of the tree, where N=0 is the nonterminal produced by the rule, N=1 is the first node matched and so forth.
- $cost[n] refers to the cost of the nth node.
- $action[n](a, b, ...) calls the action part of the nth rule. In the brackets (), one can pass parameters defined in the declaration of a terminal/nonterminal

The terminals and nonterminals used in the head of the rules have to be declared. A terminal is declared by %term terminal [14]. ICD-C provides a list of terminals for the corresponding components of the ICD-C IR. Due to this, no new terminals had to be created during this work. Nonterminals are declared with %declare <return type> nonterminal <passed arguments> [14]. The return type is a C++ type which is returned when an $action is called. Furthermore, one can define arguments which are passed in the $action call.

2.4 RISC-V

RISC-V is an instruction-set architecture (ISA). It is the fifth RISC ISA implementation from the UC Berkley and was originally intended for the sole use of research and education. However, it has found plenty of adaptation in the industry, for example with the Western Digitals "RISC-V SweRV

Core™" implementation [15]. The ISA presents the boundary between hardware and software. It defines things like the instructions set, address mode or memory layout. Common examples for ISAs are the x86 or the ARM architectures. While most personal computers at home use the x86 architecture, the large majority of smartphones rely on the ARM architecture. The main attributes of RISC-V are, that like ARM and given by its name, it follows the ideas of "reduced instruction set computing" (RISC). The idea being that instructions are as elementary as possible, making the execution of single instructions more efficient when compared to the counterpart "complex instruction set computing" (CISC). This makes it, besides other things, a good choice for embedded systems, which often have power limitations [16]. Furthermore, it is open source, which can be seen as extraordinary amongst ISAs.

Another important attribute is, that RISC-V has a customizable structure. One can choose between four base ISAs forming the foundation. Namely RV32E, RV32I, RV64I and RV128I, they differ in width of registers, address space and so on. After choosing a foundation, one can further expand the capabilities of the ISA with instruction set extensions. The important ones are M for Integer Multiplication and Division and C for 16-bit Compressed Instructions [16]. These, in addition with the base RV32I which adds basic instructions (like ADD or SLLI), make up the implemented RISC-V version in the WCC.

This implementation of RISC-V is called "RV32IMC". With the RV32I base come 32 general-purpose registers x0-x31, which are used to save arbitrary data in their 32 bits. Some of these are defined for distinct use-cases like x0 always has the value zero, or x2 holding the value of the stack pointer [16]. The registers are called general purpose because, unlike the TriCore, RISC-V does not use distinct registers for addresses. This means, whenever a pointer rule needs to generate a new register, it can call the same instruction factory function, as for example the integer rules, creating the same kind of register.

The instructions defined by the ISA make up the machine-specific code mentioned in 2.1. They describe effectively and exactly what the computational device has to execute. In RV32I there are four core groups of instructions called "instruction formats". Designated by R/I/S/U, their layout is shown in figure 2.5.

These instruction formats dictate the structure of an instruction. The R-type instructions operate on 2 registers as input and one register as their output. The I-type instructions work with immediates They have one immediate and one register as input and again have one register to save their output to. S-types are used to store the contents of registers in memory. This

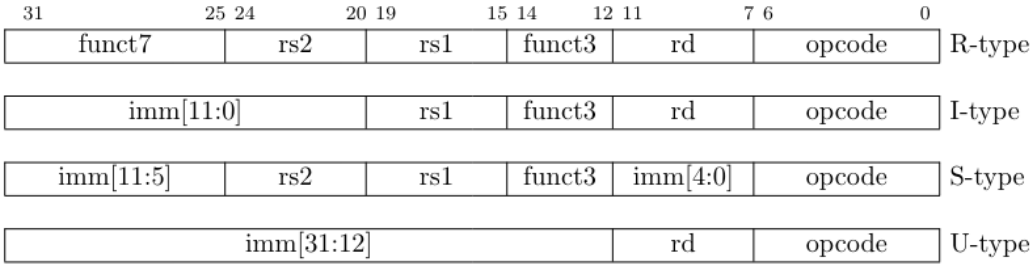

Figure 2.5: RV32I base instruction formats [16, p. 11]

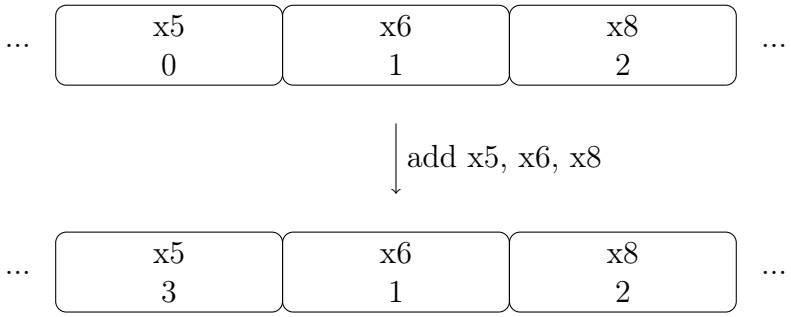

Figure 2.6: Register state for ADD instruction

is done since the information which can be held by 32 registers, all being 32-bit wide, is very limited. To extend the amount of saveable information, values can be loaded to and from memory. U-type instructions work with upper immediates. Upper immediates, in this case, refer to the first 20 bits of a register. Thus, these instructions ease the access to the upper 20 bits of a register, which are otherwise hard to manipulate.

An example of an instruction is ADD. This instruction falls into the R-type instruction format. It adds the value of a register rs1 and a register rs2 together and places the result in a third register rd. In machine-specific code, also called assembly code, ADD is executed like this:

```
add x5, x6, x8
```

Here, the registers x6 and x8 are added and the result is saved in register x5.

In figure 2.6, it is displayed how this instruction is executed. Before ADD has been executed, x5 holds the value 0, x6 holds the value 1 and x8 holds the value 2. After the execution, x5 now holds the value of 3.

The instructions, often used in chapter 3 are:

- ADDI is of I-type format. This instruction adds an immediate to the value of rs1 and saves the result in rd. The difference to the normal

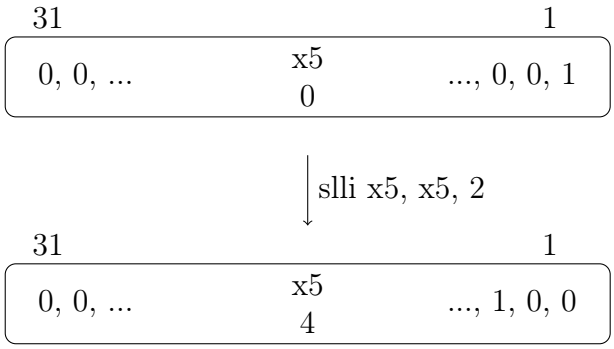

Figure 2.7: Register state for SLLI instruction

ADD thus is, that one of the values added is not saved in a register, but given directly in the assembly code. A line using ADDI might look like this:

```
addi x5, x0, 4
```

This line adds the value 4 to the value in register x0, which is always 0. The result (4) is then saved into x5.

- MUL is of R-type. MUL multiplies the value of two registers and saves the result in a third.
- SLLI is of I-type format. It shifts the bits in an register to the left. The amount of shifts is determined by the value of the 5-bit immediate and saves the result in a second register. Zeros are shifted into the lower bits. The line

  ```
  slli x5, x5, 2
  ```

  would cause the bits of register x5 to shift 2 spaces to the left, while zeros are shifted into the two lower bits. This process is exhibited in figure 2.7.
- SRLI is of I-type format. It works similarly to SLLI with the difference being, that the shift goes to the right.
- SRAI is of I-type format. It works similarly to SRLI with the difference being, that the sign bits are shifted into the upper bits, i.e., if the most significant bit was a 1, all upper bits created by the shift are also put to 1.

- MOV, this instruction shall move a value from one register to another. However, MOV is a "pseudo instruction". This indicates that it is not an actual instruction, but made up of other ones. In this case, an ADDI with 0 as the immediate, is used. This results in the value of the input register plus 0 being saved to the output register, effectively copying the value of rs1 to rd.

- LUI, (Load upper immediate) is of U-type format. The instruction is used to build 32-bit constants. It puts its upper immediate value in the top 20 bits of a register. Here the usefulness of U-Type format instructions can be seen. With an instruction like ADDI, one can place at most, a 12 bit constant into a register. Due to ADDI using the I-type format, which only has space for a 12-bit immediate. However, an ADDI together with an LUI is able to put a complete 32-bit constant into a register, for this LUI takes care of the constant's upper 20 bits, while ADDI can save the lower 12 bits to the register. [16].

- LW is of S-type. It loads a value from memory to the rd register. To specify which location in the memory should be loaded, an input register, as well as an immediate value, are used.

- SW is of S-type. This instruction stores a value from the register rs2 to a memory location. To specify which location, again, a register rs1 and an immediate value are used.

2.5 Pointers in C

The ISOAEC 9899:1999 (E) standard defines the pointer type in C as: "A pointer type may be derived from a function type, an object type, or an incomplete type, called the referenced type. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type T is sometimes called "pointer to T". The construction of a pointer type from a referenced type is called "pointer type derivation"." [17, p. 37]

To demonstrate this definition, we focus on the C code introduced earlier in section 2.2.2.1:

```
int i = 10;
int *p = &i;
```

In these two lines a pointer to the variable i is created. As per definition, the value behind the pointer p provides a reference to the variable i. This value,

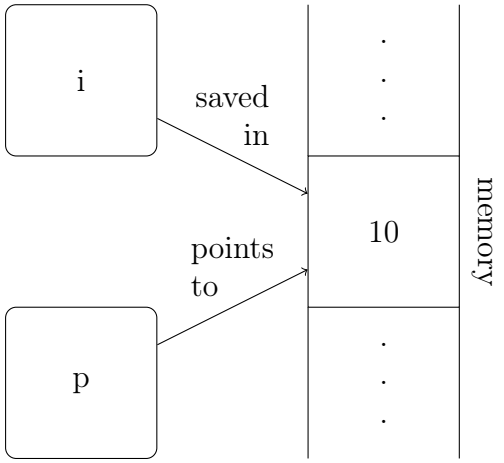

Figure 2.8: Pointer and referenced symbol relation

used for the reference, is i's address. Address in this instance refers to i's memory location. Using i's address as a reference to i effectively means, that at any point in the C code, one is able to read or modify the value behind i, by accessing the location i is stored at. To sum up this small introduction to pointers in C, the pointers allow the programmer to refer to the same space in memory, from multiple locations in the C code, which enables one to update a value in one location of the code and see the change in another one.

Going back to the example given above, the relation between p and i can be seen in 2.8.

Both symbols stand in relation to a single address in memory. This has to be represented by the assembly code, introduced in 2.4. The behavior of the C code has to be matched by the produced assembly code. Following instructions can be used to match the C code's behavior.

```
addi x5, x0, 10
sw   x5, 2(x8)
addi x6, x8, 2
```

First, the integer variable i is created. For this, two instructions introduced in 2.4 are used. This is the value of i (10) which is moved into x5 with an ADDI instruction, by utilizing the x0 register, also introduced in 2.4. Subsequently, this value is stored in the memory location, responsible for saving i. Therefore, an SW instruction is used, which saves the value in x5 to the memory location specified by: value of x8 plus 2. In the last line of the code, the pointer p is finally created. Referring back to the beginning of this section, p is expected to directly reference i, by saving i's address.

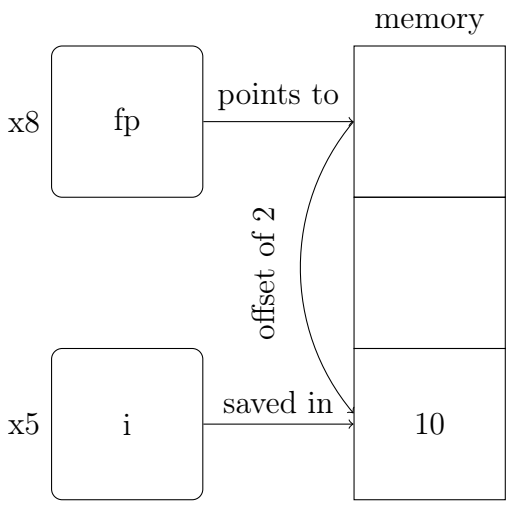

Figure 2.9: base + offset addressing

To save this address (x8 + 2), to the register x6 (responsible for saving p's value), an ADDI instruction is used. As described in 2.4, ADDI adds the immediate, which is the offset to the specified register, in this case, x8 and saves the result in x6.

The way i's address is specified here is called a base + offset address. This means, the address is made up of a base address, which provides a reference point in the memory. In this instance, the base is saved in the register x8. The reference point here, could for example be the frame pointer introduced in [5], which points to the memory address, after which all relevant data of a function is saved. The offset part of the address provides the distance of this reference point to the actual address. For a better overview, this can be seen in figure 2.9, where x8 provides the base address and the value of i lays in the address with an offset of 2. To finalize this overview of the minimal pointer example and its assembly code, in figure 2.10, the register and memory contents, after the execution of the three assembly instructions are shown. For simplicity, the address, saved in x8 was assumed to be 0. With this, it follows, that the value of i is saved in x5 and at the address 2 in memory. Since i's address is 2, x6 holding the value of p also equals 2.

The two most important operators for pointers in C have already been shown in the minimal C example. They are the address operator & and the indirection operator *. & is supposed to return the address of a symbol it is applied to. In the example that symbol was i. This return of an address can only happen if that symbol has an address (e.g., &10 does not work), if SW was never executed in the assembly code, i's value would have never been saved. These symbols with an address are called lvalue.

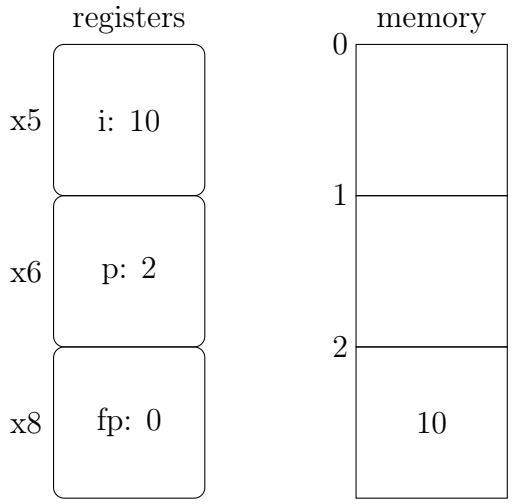

Figure 2.10: register state after executing minimal C programm

The indirection operator * dereferences the pointer it is applied to, granting access to the value the pointer is referencing.

With an expansion of the small C example upon two lines, the operation of the indirection operator can be shown closer.

```
1    int i = 10;
2    int *p = &i;
3    int j = *p;
4    *p = 11;
```

Line 3 creates a second integer variable j. The value of j is initiated with the dereferenced p. The expected outcome of these lines is, that j now also equals 10 just like i since p is a reference to i. Line 4 assigns 11 to the dereferenced p. This line is supposed to save the value 11 to i which is referenced by p. An assembly code, resulting from this C code might look like this:

```
1        addi x5, x0, 10
2        sw   x5, 2(x8)

3        addi x6, x8, 2

4        lw   x8, 0(x6)
5        sw   x5, 3(x8)

6        addi x9, x0, 11
7        sw   x9, 0(x6)
```

The code has been split up into a few sections. The first three lines remain the same. Lines 1-2 mark the creation of i. Line 3 marks the creation of p. In line 4, the first new instruction can be seen. These instructions together with line 5, create the variable j and are thus produced by dereferencing p. To load the value referenced by p, an LW instruction is used, which loads from the address saved in x6. Since x6 already points at the right location, an offset of 0 is applied. This newly created variable is then saved, just like i, however since j is a new variable, an offset of 3 is applied. Lines 6 and 7 execute the assignment of 11 to i's memory location. First, 11 is saved into the register x9 with an ADDI. The content of x9 is saved to i's location, by another SW instruction in line 7. As the address to be written to, again p's register, x6 is given, with an offset of 0. This code demonstrates the behavior of the * operator, which either needs to load or store, depending on whether it and its symbol are on the right or the left of an assignment.

In figure 2.11, the register and memory states before and after each line of assembly code can be observed. The starting state only has the fp = 0 saved. First, 10 is written to x5, then this value is also saved in memory at address 2. Afterward, this address is saved in x6. Following line 4, 10 is loaded from memory to x8. This value is then saved in memory at address 3. Line 6 places 11 into x9, followed by line 7 saving this value in memory at address 2. The three most important fields in the resulting state are the memory addresses 2 and 3 which hold i (now 11) and j (10) as well as the register x6, which holds p (2). It can observed, that the register which held i is not yet updated. This would require another load from i's address.

To conclude the introduction to c pointers in general, the two operators & and * are not the only expressions associated with pointers. One can also apply arithmetic to pointers. It is important to understand, what effect is produced, by changing the value of a pointer.

```
1    int i = 10;
2    int *p = &i;
3    i += 1;
4    p += 1;
```

This C code increments both i and p by one. The figure 2.8, introduced at the start of this chapter, would now look like the figure 2.12. This figure showcases, how in both cases the value of the symbol has been increased by one. In the case of i, this means that i now designates the value 11 however, in the case of p this means, p now points to the next location in the memory.

So far not considered is, that most data types, like integers, have a byte width greater than one. This is important to pointers since it is expected that incrementing a pointer, creates a reference to the next object in memory. The

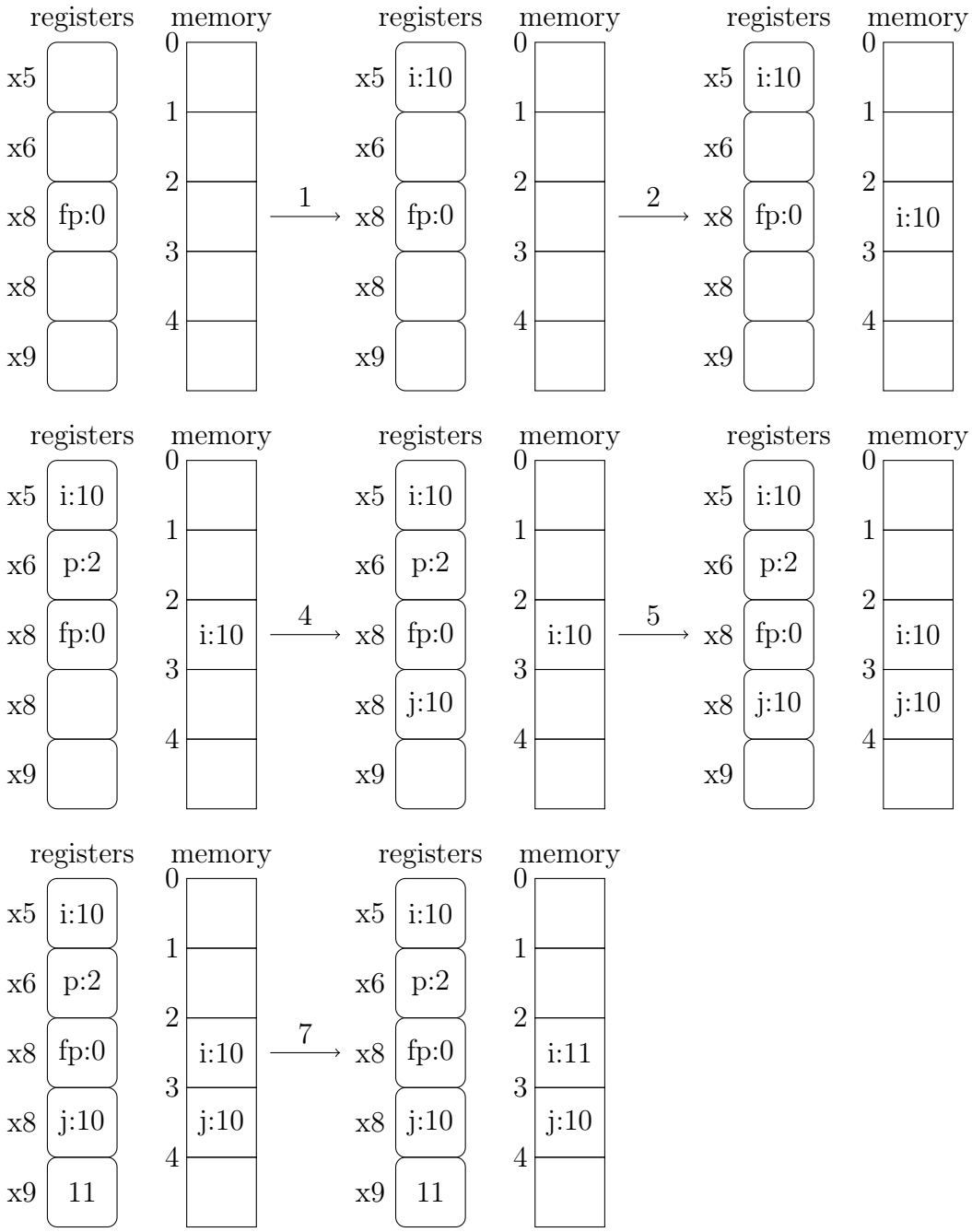

Figure 2.11: register an memory state for execution of dereferencing example

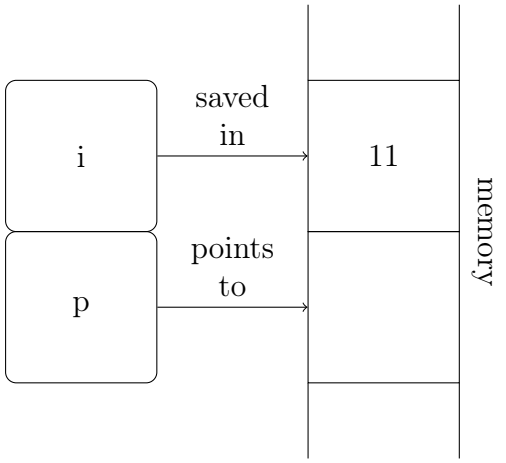

Figure 2.12: Pointer and referenced symbol relation after incrementation

problem arises with the fact that distinct locations in memory are designated by bytes. Thus, if a pointer points to an integer datatype and this pointer is then incremented by one in the C code, the actual value of the pointer needs to increase by four, as in this case, integers are 4 bytes wide.

In general, if a pointer gets incremented by a certain value n, then the actual address needs to be incremented by n times the byte width of the referenced data type. The same holds for decremented pointers as well. This behavior is shown in the assembly code representing the example for the incrementation of pointers and integers.

```
1        addi x5, x0, 10
2        sw   x5, 2(x8)

3        addi x6, x8, 2

5        addi x5, x5, 1
6        sw   x5, 2(x8)

7        addi x6, x6, 4
```

The four sections correspond to the four lines of C code. In line 5, i's register being incremented by 1, while p's register is incremented by 4 in line 7, to account for the integers byte width.

Chapter 3

Designing the Ruleset for Pointer expressions

In this chapter, the implementation of the pointer ruleset is explained. As already mentioned in section 2.2.1.2, this thesis work was in the code selection stage of the compiler. In figure 3.1, an overview of all components within this stage, which stand in correlation to the pointer ruleset, can be observed. The components which were implemented during this thesis are marked in green and the components already present are marked in blue.

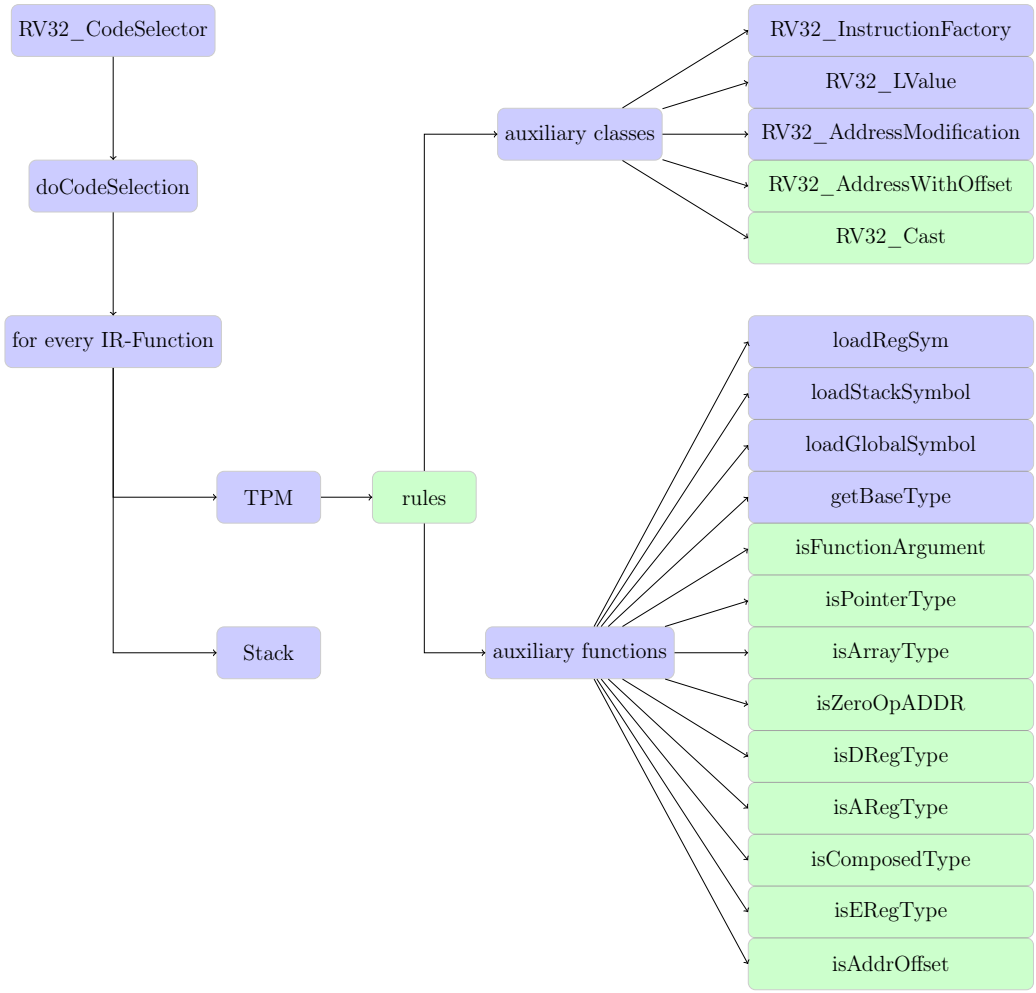

Figure 3.1: Components of the code selection stage, related to the pointer ruleset

As can be seen in figure 3.1, the whole code selection process is managed by the doCodeSelection function. Beside others, this function invokes the tree pattern matcher on the IR, which in turn invokes the matched rules. These rules rely on auxiliary classes and functions, to generate the WIR.

The doCodeSelection function and its associated components are briefly introduced in section 3.1. Since the main goal of this chapter is the introduction of the rules, the auxiliary classes and functions are introduced first to ease the explanation of rules. The auxiliary classes are explained in 3.2 and in 3.3 the auxiliary functions are shown. To lead into the declaration of the rules, first, the declared nonterminals, together with their use cases, are listed in sec. 3.4. Similarly. in sec. 3.5 the expressions which will be

matched by the rules, are listed as well. In sec. 3.6 finally, the implemented rules themselves are depicted.

3.1 Code Selector

The code selectors function was already outlined in 2.2.1.2. As can be seen in figure 3.1, it is implemented as the class RV32_CodeSelector. This class, like the other components in this chapter, lies in the /CODESEL/rv32/ folder of the WCC repository, within the header file rv32codesel.h and source code file rv32codesel.cc. The whole class was introduced in [4] and is based on the implementations in the TriCore and ARM architectures. It inherits the generic CodeSelector class, which implements many member variables and functions revolving around the central doCodeSelection function. Since this function is managing the entire code selection process, it is important to take a closer look into it.

doCodeSelection As already explained in 2.2.1.2, This function is responsible for the code generation phase.

With the goal of translating the ICD-C IR into the WIR, first, a WIR "skeleton" is generated, by iterating over the ICD-C IR structure. While doing so an equivalent WIR structure is created, with compilation units, and functions, which each get an empty basic block assigned. In consequence, doCodeSelection iterates over all files in the ICD-C. For each, the global and local symbol tables are processed, afterwards each function within the file is iterated over. Each function is then first put onto the stack, which was created in [5] and is responsible for managing objects inside the memory. Then the tree pattern matcher (TPM) is invoked. This is the crucial step in the code selection process for this thesis, since here, appropriate rules are chosen, in order to represent the ICD-C IR tree properly. These rules make up the ruleset mentioned in the title and are ultimately responsible for creating the instructions, similar to section 2.5, where C code was represented by assembly code. Once the rules have been chosen by the TPM, their action parts are invoked, which in turn make use of many auxiliary classes and functions, as can be seen in 3.1.

3.2 Auxiliary Classes

This section aims to introduce the mentioned auxiliary classes depicted in 3.1. Some of them were already briefly brought up in section 2.2.3, this section

aims to put them into perspective in the code selection process. They are used throughout the rules in section 3.6. Furthermore, there were 2 classes that had to be implemented in this thesis, in order to supplement the rules, which will be explained after the other three classes are introduced.

RV32_InstructionFactory The instruction factory class provides the rules with access to the WIR. This means it is able to insert the generated assembly code into the WIR structure. For this, the function ***createReg***, is one of the most important ones, as it creates a new RISC-V register (still virtual at this point as mentioned in 2.2.1.2) and returns a reference to it. The other functions provide the instructions included in the RISC-V RV32IMC ISA, as well as some pseudo instructions, some of which were introduced back in 2.4. These instruction-functions make up the bulk of the instruction factory, hence its name. A typical function may look like this:

```
1  void RV32_InstructionFactory::insertADD( const WIR::RV_RegV &xa,
2                                           const WIR::RV_RegV &xb,
3                                           const WIR::RV_RegV &xc,
4                                           const IR_Exp *exp,
5                                           StmtType type ) const
6  {
7    DSTART(
8      "void RV32_InstructionFactory::insertADD(const RV_RegV&, const RV_RegV
       &, "
9      "const RV_RegV&, const IR_Exp*, RV32_InstructionFactory::StmtType)
       const" );
10
11   auto &i = RV32_wirBB->pushBackInstruction(
12     { { RV32I::OpCode::ADD, RV32I::OperationFormat::RRR_1,
13         new WIR_RegisterParameter( xa, WIR_Usage::def ),
14         new WIR_RegisterParameter( xb, WIR_Usage::use ),
15         new WIR_RegisterParameter( xc, WIR_Usage::use ) } } );
16
17   ADDDEBUGINFO( i, exp, type );
18 };
```

Here the already introduced ADD instruction is inserted into a WIR basic block in line 10, with registers for in and output to be specified in lines 1-3. A function that was introduced during this thesis, is an additional ***insertADDI*** function for this class. There of course already was a version of this function implemented, as ADDI is fundamental for RISC-V assembly code. That implementation took 2 registers plus a 12 bit constant. However, an ADDI was needed, which is able to add a data label. For this purpose, a second ***insertADDI*** has been introduced, which accepts said data label.

RV32_LValue The LValue class is used to represent the lvalues mentioned in section 2.5. This means RV32_LValue represents a C symbol which designates an object and thus possesses a memory location, as well as a register.

The class inherits the generic LValue class and has member variables dedicated to save the associated address/register. The most important ones for this implementation are ***mRReg***, which points to the WIR register holding the value of the symbol. As well as ***mAddress***, which stores the register that holds the base of the memory location plus any modification that should be applied to it. The functions make access and modification to these variables possible, most notably ***storeBack***, which stores a register value back into memory, similar to the assembly examples in section 2.5. The function ***convertToBaseOffsetForm***, is used to convert the address, associated with this object, into a base offset form.

While ***convertToBaseOffsetForm*** was already present at the start of this thesis, the function body was left empty. Since some rules make use of this function, it was implemented alongside the RV32_AddressWithOffset, which is also introduced in this chapter. The function saves the address associated with the lvalue, as an address + offset, creating an object of the mentioned RV32_AddressWithOffset class along the way.

To create this RV32_AddressWithOffset object, ***convertToBaseOffsetForm*** goes ahead and invokes ***getBaseOffsetForm***. This is another function introduced during this thesis.

In case, the address already was saved as an address + offset, ***getBaseOffsetForm*** simply returns an RV32_AddressWithOffset object with that base register and that offset. If that was not the case, the function saves the entire address, which is to be converted, in a newly created register with an LUI instruction for the upper 20 bits of the address and with an additional ADDI instruction for the lower 12 bits. With the created register and an offset of zero, the RV32_AddressWithOffset object is returned.

RV32_AddressModification The AddressModification class represents an address register that has a pending modification. This effectively means that objects of this class represent pointers that are designated to be changed, however, this change has not yet happened. The class is inherited from the generic AddressModification class.

For a proper representation, the register of the pointer and the referenced data type are stored in member variables. For the modification, either an integer or a register offset is stored, together with the write-back policy and whether the offset is to be added or subtracted. Important member variables are:

- ***mAReg***, points to the WIR address register to be modified.
- ***mBaseLValue*** points to an lvalue, this is used instead of ***mAReg*** if the address is also stored in the memory.

- ***mOffset*** stores the offset of an address modification (relative to the base).
- ***mBaseType*** points to the base type of the pointer/address, i.e., the referenced datatype.
- ***mOReg*** points to an WIR register containing the offset.
- ***mModOper*** specifies whether the address modification is to be added or subtracted from the original value.

The AddressModification class also has functions to perform the pending modification and to grant access to various member variables. Three of these functions are important for this thesis.

- ***applyModification***, inserts instructions to perform the pending modification to the address. Going back to the very last example in section 2.5, this function would be responsible to increment the address by four, in order to point to the next integer.
- ***createLoad*** generates instructions, which load the value from the address encoded by the AddressModification object, into a specified register.
- ***createStore*** generates instructions, which store the specified register to the address encoded by the AddressModification object.

RV32_AddressWithOffset The first class implemented in this thesis is the AddressWithOffset class. It is used to represent the base + offset address, introduced in 2.5 and is derived from a generic AddressWithOffset Class. To represent the address, the class has two member variables. ***mAReg***, which points to the register holding the base address. And ***mOffset*** which stores the offset in bytes. The functions ***getAReg*** and ***getOffset*** grant access to the variables by simply returning them.

RV32_Cast The second class implemented, is the RV32_Cast class, which also does already exist in the other two architectures within the WCC. The class is used to perform casts. Many tests performed during this thesis required casting of some kind, which necessitated the creation of RV32_Cast, however, the class was only implemented to a point, where all casts necessary for the pointer ruleset, were operational. In consequence, parts of this class still need implementation to actually support all kinds of casts. The

class thus far has two important functions. The ***doCasting*** function and ***doTruncation***.

doCasting is the crucial method as it is responsible to perform the type conversion between the source type and the destination type, to be cast to. In this function first, the source and the destination data types are checked. There are 10 possible cases of source and destination datatype combinations.

1. Casting of integral types to integral types or Bool.
2. Casting of real types to integral types.
3. Casting of integral types to real types.
4. Casting of real types to real types.
5. Casting to void.
6. Casting to pointers.
7. Casting from pointers.
8. Casting between pointers.
9. Casting real types to Bool.
10. Casting pointers to Bool.

In the list above integral types are: unsigned long long, long long, unsigned long, long, unsigned int, int, unsigned short, short, unsigned char, char, bool.

And real types are: long double, double, float.

Each case needs to be handled differently, which necessitates the use of if-else statements. For this thesis, 5 of those cases were important and thus implemented, namely cases 1,5,6,7,8.

- The first one handles casting between integral types. For this instance, the second implemented function of this class is called, namely ***doTruncation***.
- The case handling a cast to void just returns a newly created and thus empty register.
- For casting from or to pointers, a single MOV instruction moves the value from source to destination.
- For casting between pointers nothing is to be done.

The ***doTruncation*** function is responsible for the truncation of a register's contents to the bit width of the associated data type. This is needed since not all data types have the same bit width.

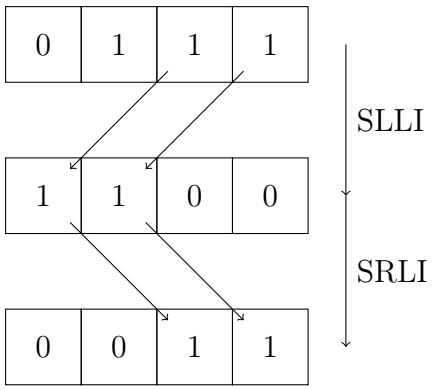

Figure 3.2: truncation of bits

If the destination datatype is a char, unsigned char, short, or unsigned short, a MOV instruction followed by a left shift SLLI and a right shift SRLI (or SRAI for twos complement)are inserted. All of these instructions have been introduced in 2.4. For these data types the bit width actually was shorter, because of this, the unwanted bits are truncated, by shifting them out of the register with an SLLI, then SRLI/SRAI is used to shift the bits back into their original position. For better understanding, this shifting process is shown in figure 3.2.

If the destination datatype is not one of the ones mentioned above, a single MOV instruction is inserted.

3.3 Auxiliary Functions

This section aims to introduce the mentioned auxiliary functions depicted in 3.1. These functions are used throughout the rules in section 3.6. First, three functions, which were already implemented are explained. Afterward, the functions implemented in this thesis are discussed.

Load symbols There are three functions responsible for loading symbols into their registers. They are found in the same folder as all of the code selector related source code, within the file rv32_prolog.m4.

- ***loadRegSym*** loads a non-stack, non-global symbol into its appropriate register.

- ***loadStackSymbol*** loads a local stack symbol into its appropriate register.
- ***loadGlobalSymbol*** loads a global symbol into its appropriate register.

loadRegSym and loadStackSymbol showcase a feature of the WCC. This feature is the distinction between local symbols residing in registers only and local symbols additionally residing on the stack, i.e., in memory. A symbol is automatically classified as a register symbol, if it does not need to be saved in memory, by the WCC. This distinction helps to keep symbols out of the memory, for which being stored in a register is sufficient, subsequently reducing memory accesses [12]. Because of this loadRegSym gets away with simply saving a value into a register, while loadStackSymbol, as well as loadGlobalSymbol, have to work with LW instructions to load their symbols values from their associated memory address.

getBaseType The function ***getBaseType*** can again be found in the rv32-_prolog.m4 file. As its name suggests, it is able to return the data type referenced by a pointer. For this, the function makes use of the ICD-C IR, which saves the referenced datatype.

computeSizeOf The function ***computeSizeOf*** is used to calculate the byte width of a data type. It is also declared in the rv32-_prolog.m4 file and was already present at the start of this thesis.

Is-Functions These functions are mostly used in the costs part of rules and make use of the ICD-C IR to check for certain attributes of terminals/nonterminals. All of them are found in the rv32_prolog.m4 file. To show an example of how these functions operate, the arguably most important function, ***isPointerType***, checks the result of getType() against IR_Type::POINTER and returns the created bool. The resulting code looks as follows: (For wrongly generated type IDs a further check is also performed)

```
1  bool isPointerType( const IR_Type &t )
2  {
3    DSTART( BOOST_CURRENT_FUNCTION );
4
5    return(
6      ( t.getType() == IR_Type::POINTER ) ||
7      // The IR produces wrong type objects with invalid type IDs from time
       to
8      // time. So, we must try to cast the object as well, because in those
       cases,
9      // this is the only thing that works.
10     ( dynamic_cast<const IR_PointerType*>( &t ) &&
11           !dynamic_cast<const IR_ArrayType*>(  &t ) ) );
12 };
```

Further functions added are listed below, these all return true if the case they check against is actually true:

- ***isFunctionArgument*** checks, whether the given expression is a function argument symbol.
- ***isArrayType*** checks, whether the given type is an array type
- ***isZeroOpADDR*** checks, whether the given expression is an address operator expression that must be handled without emitting any code, like &*a or &a[i].
- ***isDRegType*** checks, whether the given type is a void, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, or bool.
- ***isARegType*** checks, whether the given type is a pointer or array type.
- ***isComposedType*** checks, whether the given type is a struct or a union.
- ***isERegType*** checks, whether the given type is a double, long double, long long, or an unsigned long long type.
- ***isAddrOffset*** checks, whether the given expression can be represented by the addrOffset nonterminal, introduced in 3.4.

3.4 Nonterminals

As could be seen in section 2.3, nonterminals play a crucial role in the ruleset. To use nonterminals in the rules, they first have to be declared. This section aims, to list every nonterminal that was declared during this thesis. A few nonterminals will be listed in table 3.3 and introduced in section 3.4.1. These were already present at the start of this thesis, but are nonetheless important to the ruleset. An overview of the introduced nonterminals can be seen in table 3.4. Each nonterminal directly belonging to the pointer ruleset is going to be looked at separately in section 3.4.2, whereas related nonterminals are explained in 3.4.3. This section focuses on the rule component visualized in figure 3.1.

3.4.1 Existing Nonterminals

These nonterminal where already present, they can be found in rv32_decl.m4.

Returntype	Nonterminal	Parameter	Describtion
RegV	reg		data register
LValue	deref_reg	bool loadResult	data register + memory location
void	stmt		statement

Figure 3.3: nonterminals which where already present in the WCC

reg and deref_reg The reg nonterminal symbolizes a register containing data, like an integer. Similarly, deref_reg also refers to a data register, but this time the object also has an associated memory location, in order to save back possible modifications to that memory location. The parameter loadResult is used to signal the called rules whether the result actually should be loaded from memory. In their declaration:

```
%declare<WIR::RV_RegV &><RV32::dummyRegV> reg;
%declare<RV32::RV32_LValue> deref_reg<bool loadResult>;
```

the use of the lvalue class can be seen. While the normal reg nonterminal is simply represented by a register, the deref_reg nonterminal is represented by an lvalue, indicating the possession of a register as well as a memory location.

stmt The stmt nonterminal symbolizes an entire statement and thus usually is the last terminal left after tree pattern matching. Because of this, nothing is to be returned.

```
%declare<void> stmt;
```

3.4.2 Introduced Nonterminals

These terminals were declared as part of this thesis. They lie in the rv32_pointer_decl.m4 file.

areg and deref_areg The areg nonterminal symbolizes a register containing an address, i.e., a pointer. Similarly, deref_areg also refers to an address register, but again, the object also has an associated memory location in order to save back possible modifications to that memory location. The parameter loadResult is used to signal the called rules whether the result actually should be loaded from memory (if not just the address is computed). It is important that the distinction between data registers and address registers is only made in code, to ease the distinction between rules for other data types from the ones performing pointer related translation.

Return type	Nonterminal	Parameter	Description
RegV	areg		address register
LValue	deref_areg	bool loadResult	address register + memory location
AddressModification	modified_areg		areg with pending modification
IR_Integer	constAddress		constant address in memmory
RegV	zero_op_areg		& * e shall never emit code
IR_Integer	addrOffset		integer address offset
RegV	implicit_castable_reg		register to be casted (implicit)
RegV	explicit_castable_reg		register to be casted (explicit)
WriteBackInfo	ca_result_areg		result address register of assignment
WriteBackInfo	ca_lhs_areg	bool loadResult	left hand side of assignment

Figure 3.4: nonterminals added during this thesis

```
%declare<WIR::RV_RegV &><RV32::dummyRegV> areg;
%declare<RV32::RV32_LValue> deref_areg<bool loadResult>;
```

The lvalue class is used for the deref_ version of the areg nonterminal.

modified_areg This nonterminal symbolizes an address register to be modified. However, no instructions have been generated yet to perform said modification. For this, the return type is set to use an AddressModification object, introduced in 3.2. At a later stage, the AddressModification object will then be able to insert the instructions for the modification with its applyModification function.

```
%declare<RV32::RV32_AddressModification> modified_areg<void>;
```

constAddress The constAddress nonterminal is an integer representing an address in memory.

```
%declare<IR_Integer> constAddress;
```

zero_op_areg The C standard defines that the dereference operator, followed by the address operator, shall not produce any assembly code [17, sec. 6.5.3.2]. This circumstance is represented by the zero_op_areg. While no assembly code shall be generated, the symbol the operators were applied to and its associated register still needs to persist, for this reason, zero_op_areg returns the register:

```
%declare<WIR::RV_RegV &><RV32::dummyRegV> zero_op_areg<void>;
```

addrOffset addrOffset symbolizes the offset of a base + offset address.

```
%declare<IR_Integer> addrOffset;
```

This nonterminal might seem very similar to the constAddress nonterminal. However, the constAddress represents an entire address, while addrOffset only represents the offset part of an address.

3.4.3 Non Pointer Ruleset Nonterminals

Unlike the nonterminals before, the ones following do not directly fit into the pointer ruleset category. However, these nonterminals are crucial for numerous tests and usually still are strongly related to pointers. Thus they were also introduced.

implicit_castable_reg and explicit_castable_reg These nonterminals represent a register that shall be casted in a future rule to its target type. implicit_castable_reg is used to represent implicit casts, like *double d = 10;*. explicit_castable_reg represents explicit casts, like *int i = (int)10.0;*.

```
%declare<WIR::RV_RegV &><RV32::dummyRegV> implicit_castable_reg;
%declare<WIR::RV_RegV &><RV32::dummyRegV> explicit_castable_reg;
```

ca_lhs_areg and ca_result_areg Ca_lhs_areg is used to represent the left-hand side of an assignment, with a register, holding an address. Ca_result_areg is used to represent the result of an assignment, again with a register holding an address. These nonterminals start off with *Ca_* indicating a relation to casting rules. This is done since there are two possible casts associated with an assignment. Considering the C code example:

```
1    int i;
2    float f;
3    i+= f;
```

For line 3, i is cast to a float. The result of the addition between f and the casted i must then be cast back to int. Ca_lhs_areg thus represents the first possible cast and Ca_result_areg represents the second one.

```
%declare<RV32::RV32_WriteBackInfo> ca_result_areg<void>;
%declare<RV32::RV32_WriteBackInfo> ca_lhs_areg<bool loadResult>;
```

The loadResult parameter again is used to signal the later called rules, whether the result actually should be loaded from memory. The RV32_-WriteBackInfo struct provides information about a potential memory write-back, for this, it holds an RV32_LValue, a bool denoting whether a write-back actually has to be done or not and an additional register to which intermediate results shall be written.

3.5 Handled Expressions

The main goal of this work is to create a code-selector ruleset for pointer expressions. Each of these expressions must be represented by a terminal. Important expressions and their matching terminals are explained below, and listed in table 3.1. Each expression also has an example, here p and q are pointers of type *int while i is of type int. The examples are not always completely covered by a single terminal. All of these terminals were already declared at the start of this thesis.

SymbolExp The tpm_SymbolExp is generally used to load or store symbols. To work with pointer symbols, they of course have to be loaded as well. While this expression is important for about every data type, as it generally is used for every kind of symbol, it of course plays a fundamental role for pointers and specifically working with pointer symbols as well.

UnaryExpDEREF Next up are the two terminals representing the most fundamental operators for using pointers in C. Starting with the indirection operator *, which is represented by tpm_UnaryExpDEREF. As already shown in 2.5, its purpose is to provide access to the value behind the address of a pointer.

Expression	Terminal	Example
Loading pointer symbols	tpm_SymbolExp	p
Indirection operator	tpm_UnaryExpDEREF	$*p$
Address operator	tpm_UnaryExpADDR	$\&i$
Addition	tpm_BinaryExpPLUS	$p + i$
Subtraction	tpm_BinaryExpMINUS	$p - i$
Pointer subtraction	tpm_BinaryExpMINUS	$p - q$
Postincrement	tpm_UnaryExpPOSTINC	$p + +$
Postdecrement	tpm_UnaryExpPOSTDEC	$p - -$
Preincrement	tpm_UnaryExpPREINC	$+ + p$
Predecrement	tpm_UnaryExpPREDEC	$- - p$
Pointer assignment	tpm_AssignExpASSIGN	$p = q$
Addition assignment	tpm_AssignExpPLUS	$p+ = i$
Subtraction assignment	tpm_AssignExpMINUS	$p- = i$

Table 3.1: Handled Expressions

UnaryExpADDR The address operator &, which is represented by tpm_-UnaryExpADDR, needs to retrieve the address of the symbol it is applied to. In 2.5 it was mentioned, that the address operator only works for operands that are able to have an address, i.e., are an lvalue. The expression &2022 thus does not work, as the number 2022 is not an object by itself. This effectively means that the associated nonterminal needs to be of the deref_[...] type to actually have an associated memory address.

BinaryExpPLUS and BinaryExpMINUS These are arithmetic operations. The natural starting points are the addition and subtraction of a pointer with an integer. For this tpm_BinaryExpPLUS as well as tpm_-BinaryExpMINUS are used. These generally accept many kinds of different data types and are thus not exclusive to pointers. However, once a pointer-type symbol is involved, the additions or subtractions are altered a bit. As already noted in section 2.5, pointers are generally incremented/decremented by multiples of their referenced data type byte width, in order to iterate the pointer object wise. Figure 3.5 shows an example, where 2 objects of int type lay beside each other in memory. The pointer points to the first int object. If the integer 1 is added to said pointer it is expected to now point to the second int in memory. If one would not take the byte width into account, the pointer would instead just point to the beginning of the second byte of the first integer.

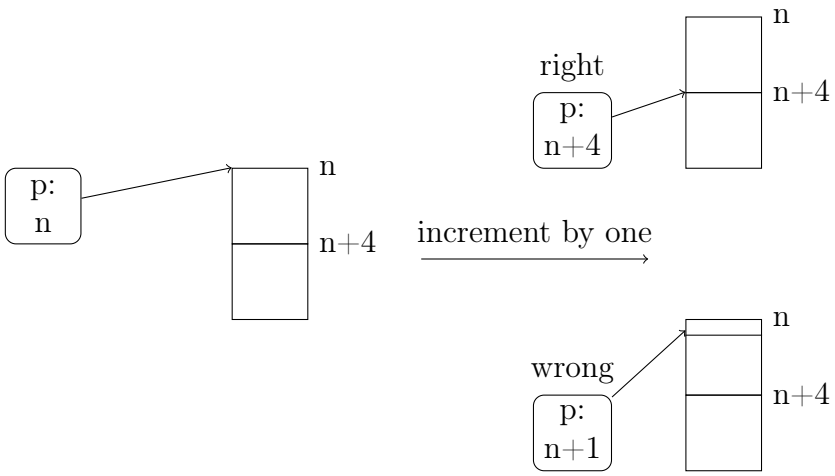

Figure 3.5: pointer incrementation

Pointer to pointer subtraction (tpm_BinaryExpMINUS), similarly does not result in the number of bytes in between the two pointers, but the number of objects, with the width of the data type, pointed to. For this reason, the two subtracted pointers have to point to the same data type. The addition between 2 pointers does not exist as this does not have a use case.

UnaryExpPOSTINC and UnaryExpPOSTDEC The terminals tpm_UnaryExpPOSTINC and tpm_UnaryExpPOSTDEC add or subtract the value 1 to a pointer, again this actually adds or subtracts the byte width from or to the pointer as explained above. However, the operation itself returns the unmodified value of the symbol. To showcase this:

```
1    int i = 10;
2    int *p = &i;
3    int *q;
4    q = p++;
```

in line 4 the post-incrementation of the pointer p is performed. Recently p was pointing to the address of i. If we assume that i had the address n, we now would expect the pointer q to point at n, while p now is supposed to point to n+4.

UnaryExpPREINC and UnaryExpPREDEC tpm_UnaryExpPREINC and tpm_UnaryExpPREDEC also increment or decrement the value of a pointer by 1. The exception to their postinc. or postdec counterparts is, that the modified value is returned. So for:

```
1    int i = 10;
2    int *p = &i;
3    int *q;
4    q = ++p;
```

both q and p are now supposed to point to the address n+4.

AssignExpASSIGN The assignment of pointers is covered with the tpm_AssignExpASSIGN terminal. For this case of assigning an address to a pointer, the address of the right-hand side is simply written to the operand of the left-hand side. C also allows for assigning and adding/subtracting at the same time. This is covered by tpm_AssignExpPLUS and tpm_AssignExpMINUS. These take the address of the left-hand side, added, or subtracted with the right-hand side and then written back to the left-hand side.

3.6 OLIVE Rules

With this section, we arrive at the heart of the rule component seen in figure 3.1. During this thesis, many rules were introduced, the aim of this section is to take a look at these rules and explain their inner workings. Not all of them directly belong to pointer expressions, however, all were necessary in order to successfully compile the tests in the testbench. Due to the amount of more than sixty rules created, the rules' source code will not be shown for each rule, instead, the characteristics of each rule will be discussed and examples will be given along the way. All important rules are also depicted in figures, which show the tree pattern they match against and what nonterminal they return.

3.6.1 Loading Pointer Symbols

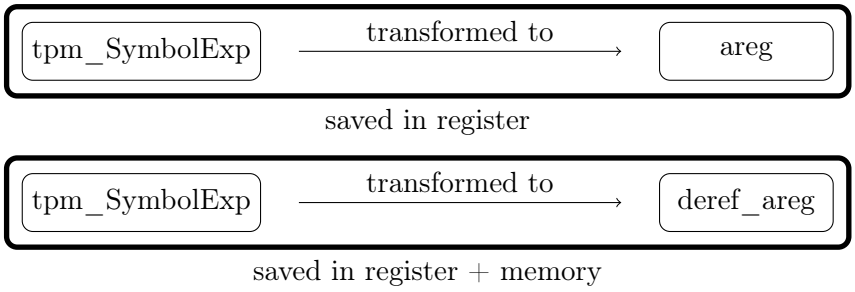

Figure 3.6: Loading pointer Symbols.

The two rules depicted in figure 3.6 handle the loading of pointer symbols. They transform a tpm_SymbolExp terminal, introduced in 3.1, into either an areg or deref_areg nonterminal, which were introduced in 3.4.

areg: tpm_SymbolExp is applied if the loaded address does not need to be saved in the memory. Instead, it is sufficient for this rule to load the address into a register. There are 3 versions of the ***areg: tpm_SymbolExp*** rule. These can be described by:

1. version one is for for the case that the symbol is of pointer type, is not global and does not reside in the stack. It calls the ***loadRegSym*** function and then returns the register object created by the function.

2. Version two handles function symbols. In a similar fashion to version one, it calls the already existing ***loadGlobalSymbol*** function and subsequently returns the register taken from the created lvalue object.

3. Version three is for the case that the symbol is of array type and explicitly not a function argument.It makes a distinction and checks whether the symbol is global, then ***loadGlobalSymbol*** is used, if not and the symbol is also not on the stack ***loadRegSym*** is used, else the already existing ***loadStackSymbol*** function is used.

Different versions of the same rule are sometimes needed, to distinguish different cases regarding the terminal that is to be transformed. To ensure the right rule is taken, the cost part of the rules, introduced in 2.3, enforce their specific case. If they are supposed to be used the cost function will reflect the proper cost, if they are not supposed to match the case at hand, the cost will be set to infinity. This is done by using the *COST_INFINITY* expression. The source code for the first version of ***areg: tpm_SymbolExp*** reflects this behavior in its cost section:

```
1  areg: tpm_SymbolExp
2  {
3    // Acquire the expression and the symbol itself.
4    auto &symExp = dynamic_cast<IR_SymbolExp &>( *$1->getExp() );
5    auto &sym = symExp.getSymbol();
6
7    // Match this rule iff the symbol is of pointer type, is not global and
       does
8    // not reside in the stack.
9    if ( ( RV32::isPointerType( sym ) ||
10        ( RV32::isArrayType( $1->getExp()->getType() ) &&
11        RV32::isFunctionArgument( *$1->getExp() ) ) ) &&
12        !sym.isGlobal() && ( RVCODESEL.getStack().getSymbolOffset( sym ) < 0
       ) )
13     $cost[0] = RV32::loadRegisterSymbolCost( symExp );
14   else
15     $cost[0] = COST_INFINITY;
16 }
```

```
17 =
18 {
19   RV32::DEBUG_RULE_ACTION( "areg: tpm_SymbolExp", $1 );
20
21   // Acquire the expression.
22   auto &symExp = dynamic_cast<IR_SymbolExp &>( *$1->getExp() );
23
24   // And load the symbol.
25   return( dynamic_cast<RV_RegV &>( RV32::loadRegSym( symExp ) ) );
26 };
```

In the cost section of the rule, many of the introduced is-functions from section 3.3 are used. In lines 9-12 an if statement checks, whether the symbol is a pointer, or of array type. Furthermore, the if statement ensures, that no global or stack symbols are matched by this rule. If any of these requirements are not fulfilled, the cost of this rule is set to infinity (line 15), which makes it unusable. Line 13 shows how costs for these kinds of functions are calculated. Usually, the functions have a cost counterpart, which adds up the cost for possibly added instructions and returns it.

deref_ areg: tpm_ SymbolExp handles the case, where the address does need to be saved in memory. It has 2 distinct versions. The first version is for the case that the symbol is of pointer type, is not global and does reside in the stack. Here, ***loadStackSymbol*** is used. The second version handles global symbols, where subsequently ***loadGlobalSymbol*** is used to load the pointer symbol.

3.6.2 The Indirection Operator *

Next are the rules representing the indirection operator (*). The three rules, which represent this operator can be seen in figure 3.7. It can be seen, that these do not produce any non-deref_ nonterminal like reg or areg. This is the case, because dereferencing a pointer, necessitates that the symbol pointed at, was an lvalue. Because of this, dereferencing a pointer will always return an lvalue.

deref_ areg: tpm_ UnaryExpDEREF(areg) is used in the case that the data type produced by dereferencing a pointer is again of pointer type and thus again returns an address. The rules source code looks as follows:

```
1  deref_areg: tpm_UnaryExpDEREF( areg )
2  {
3    auto &t = $1->getExp()->getType();
4    if ( RV32::isPointerType( t ) &&
5         !RV32::isFunctionPointer( *$2->getExp() ) )
6      $cost[0] = $cost[2] + RV32I::OperationFormat::RC12R_1.getSize();
7    else
8      $cost[0] = COST_INFINITY;
9  }
10 =
11 {
```

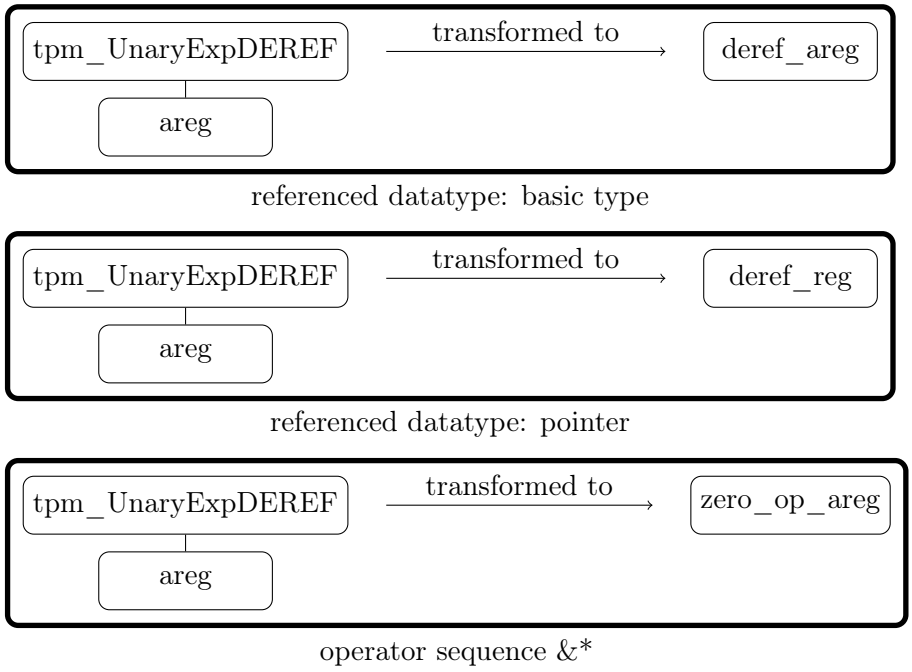

Figure 3.7: Rules representing the the indirection operator *.

```
12  RV32::DEBUG_RULE_ACTION( "deref_areg: tpm_UnaryExpDEREF( areg )", $1 );
13
14  auto &t = $1->getExp()->getType();
15  auto &p = $action[2]();
16
17  RV_RegV &r = RVINSTRUCTIONS.createReg();
18
19  if ( loadResult ) {
20
21    RVINSTRUCTIONS.insertLW( r, 0, p, $1->getExp() );
22  }
23
24  return(
25    RV32::RV32_LValue {
26      &r, RV32::RV32_AddressModification {
27        p, 0, &t, true } } );
28 };
```

First, a new register is created in line 17. Then, in case loadResult is true, an LW instruction is added which loads from the address of the areg child node into the newly created register. Then with the register and with a newly created AddressModification object, an lvalue object is created and returned in lines 24-27. The cost part ensures that the expression is of pointer type in line 4. Furthermore, the address this is applied to, must not be a function symbol (line 5). If these two things hold true, the cost of the possibly used LW instructions is added in line 6.

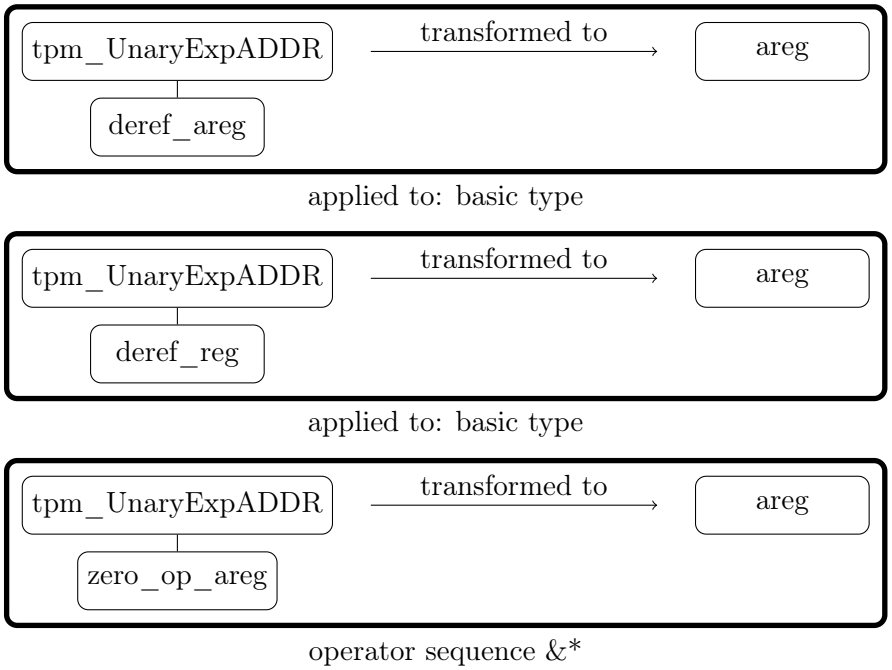

Figure 3.8: Rules representing the address operator &.

deref_reg: tpm_UnaryExpDEREF(areg) is applied when the type at the address of the pointer is a normal data type. Again, a new register is created and afterward, an AddressModification object is created with the areg register. If loadResult is passed as true, the ***createLoad*** function is called on the newly created AddressModification. For the return, an lvalue object is created with the register and the AddressModification object, as seen in the rule above.

The third rule ***zero_op_areg: tpm_UnaryExpDEREF(areg)*** is for the special case that the operators &* are applied to a symbol. In this case, as mentioned earlier, no extra instruction is to be emitted. Thus this rule just returns the register of the areg nonterminal. The costs do not enforce anything here and also just take the value from their sub-tree as there is nothing to be done that could add costs.

3.6.3 The Address Operator &

The rules representing the address operator can be seen in figure 3.8. All of them produce an areg nonterminal which is to be expected from the address operator. They differ however in the child nodes of the terminals.

The first one ***areg: tpm_UnaryExpADDR(deref_areg)*** matches

for a deref_areg. this rule is used in cases where one wants the address of a pointer, i.e., creating a pointer to a pointer. The second one ***areg: tpm_ UnaryExpADDR(deref_ reg)*** does match against a deref_reg nonterminal as the child node. This is used for the cases where the address, of a normal data type object like int, is to be retrieved, resulting in a pointer to the specific datatype (for example int).

Both rules have the exact same cost and action parts, which can be seen in the following code snippet:

```
1  areg: tpm_UnaryExpADDR( deref_areg )
2  {
3    if ( !RV32::isZeroOpADDR( *$1->getExp() ) )
4      $cost[0] = $cost[2] + RV32I::OperationFormat::RRC12_1.getSize();
5    else
6      $cost[0] = COST_INFINITY;
7  }
8  =
9  {
10   RV32::DEBUG_RULE_ACTION( "areg: tpm_UnaryExpADDR( deref_areg )", $1 );
11
12   auto lvalue = $action[2]( false );
13   lvalue.convertToBaseOffsetForm( $1->getExp() );
14
15
16   auto &r = RVINSTRUCTIONS.createReg();
17
18   if ( lvalue.getOffset() == 0 )
19     RVINSTRUCTIONS.insertCMV(
20       r, dynamic_cast<RV_RegV &>( lvalue.getAddress().getAReg() ),
21       $1->getExp() );
22   else
23     RVINSTRUCTIONS.insertADDI(
24       r, dynamic_cast<RV_RegV &>( lvalue.getAddress().getAReg() ),
25       lvalue.getOffset(), $1->getExp() );
26
27   return( r );
28 };
```

The costs for both rules first ensure that the deref_areg or deref_reg are not of zero_op_address type (&*) in line 3. If that is not the case, the actual cost for the subtree and instruction is added. Otherwise, the cost is set to infinity. In their action part both rules, first retrieve the lvalue object behind deref_areg/deref_reg in line 12. On this object, the ***convertToBaseOffsetForm*** function is called in line 13. This ensures, that the address of the object now is a base + offset address. Then a new register is created and depending on whether the address has a nonzero offset or not, the address of the symbol is simply moved from its old register into the newly created one. Otherwise it is added up with the offset and subsequently the result is placed into the new register.

The third rule seen in figure 3.8, ***areg: tpm_ UnaryExpADDR(zero-_op_ areg)***, again applies to the case where the operators & and * were used in this succession. With this rule, the path such an expression takes

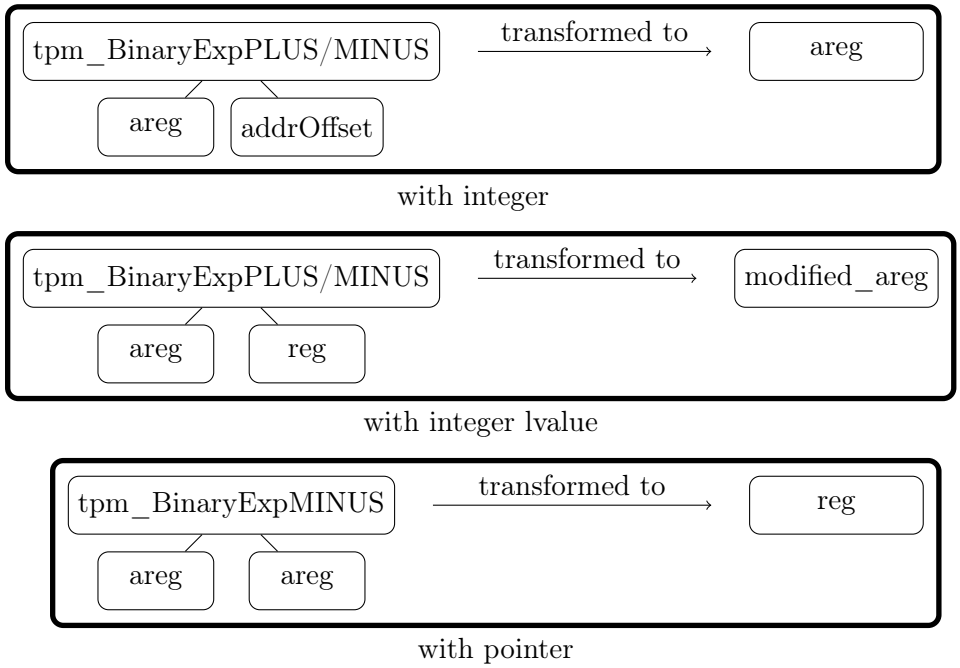

Figure 3.9: Rules representing the addition and subtraction of pointers.

can be seen. First, a pointer p gets the * operator applied to it, which gets matched by ***zero_op_areg: tpm_UnaryExpDEREF(areg)*** introduced in section 3.6.2. Afterwards, the & operator is applied to *p. This rule matches against that and returns the areg, i.e., the pointer, without generating any assembly code whatsoever.

3.6.4 Addition and Subtraction

In this subsection, rules for addition and subtraction with pointers are introduced. The figure 3.9 compresses the tpm_BinaryExpPLUS and the tpm_BinaryExpMINUS into a single node to keep the figure tidy.

For the addition of an address with an integer, there exist two rules. ***areg: tpm_BinaryExpPLUS(areg, addrOffset)*** covers the case where an integer, not directly representing an object (i.e., an lvalue in the sense of C), is part of the addition. As an example, if the C code had a

```
p+10
```

expression, this rule could be used. The rule matches against the areg to be incremented, the addrOffset representing the added integer and the tpm_BinaryExpPLUS terminal representing the +.

```
1  areg: tpm_BinaryExpPLUS( areg, addrOffset )
2  {
3    $cost[0] = $cost[2] + $cost[3] + RV32I::OperationFormat::RRC12_1.getSize()
       ;
4  }
5  =
6  {
7    RV32::DEBUG_RULE_ACTION( "areg: tpm_BinaryExpPLUS( areg, addrOffset )", $1
       );
8
9    auto *t = RV32::getBaseType( $2->getExp()->getType() );
10   const int off = $action[3]().getIntValue() * RV32::computeSizeOf( t );
11
12   auto &p = $action[2]();
13
14   auto &r = RVINSTRUCTIONS.createReg();
15   RVINSTRUCTIONS.insertADDI( r, p, off, $1->getExp() );
16
17   return( r );
18 };
```

As already discussed in section 3.5, first, the actual byte offset needs to be calculated. This is done by multiplying the value of the integer, represented by the addrOffset nonterminal, with the byte width of the base datatype, in line 10. To retrieve the integer value ***getIntValue*** is called on the addrOffset object. For the byte width of the base type, ***computeSizeOf*** is called on the type retrieved by the ***getBaseType*** function. After calculating the byte offset, a new register is created in line 14. This register is used as the target register for the addition of the address offset and the actual address register behind the areg nonterminal. This addition is carried out by an ADDI instruction, the instruction is inserted in line 15. The newly created register can then be returned.

The second rule ***modified_areg: tpm_BinaryExpPLUS(areg, reg)*** is used in case the integer to be added to an address actually designates an object (i.e., an lvalue in the sense of C) and thus resides in a register. An example of this would be

```
p+i
```

where i is an int type. Since the modified_areg is used as a nonterminal here, the rule actually does not have to create assembly code, as this can be done at a later stage. However, all relevant information about the modification of the address has to be provided to the AddressModification object.

```
1  // Retrieve the base type.
2    auto &ptrType = dynamic_cast<IR_PointerType &>( $2->getExp()->getType() );
3    auto &t = ptrType.getBaseType();
4
5    // Evaluate both sides of the expression, left-hand side first.
6    auto &lhsReg = $action[2]();
7    auto &rhsReg = $action[3]();
8
```

```
9   // Assemble the RV32_AddressModification.
10  return(
11    RV32::RV32_AddressModification {
12      lhsReg, rhsReg, &t, AddressModification::ModTime::NONE,
13      AddressModification::ModOper::ADD } );
```

To provide said information, the base datatype of the pointer is retrieved via getBaseType in line 3. Then the registers of the areg and reg are retrieved. With this information, in lines 11-13, an AddressModification object is created. In line 13 it is also specified that this modification is an addition. Since no instructions are added here, the costs just represent the costs of the sub-tree.

The subtraction rules for address plus integer subtraction work the same way as their addition counterparts. The small change is, that for ***areg: tpm_BinaryExpMINUS(areg, addrOffset)*** the calculated address offset is turned into its negative. For the rules source code, this means that line 10 of ***areg: tpm_BinaryExpPLUS(areg, addrOffset)*** changes into:

```
1 const int off = -( $action[3]().getIntValue() * RV32::computeSizeOf( t ) );
```

For ***modified_areg: tpm_BinaryExpMINUS(areg, reg)*** the AddressModification constructor is called with ModOper::SUB as an argument, symbolizing that the integer shall be subtracted.

As already mentioned earlier, for the subtraction terminal, there is an additional case for the subtraction of two addresses with each other. The rule ***reg: tpm_BinaryExpMINUS(areg, areg)*** takes care of this. In order to possibly reduce the computational complexity, the rule starts with checking whether the byte size of the base datatype is a power of two and computing this power. In the next step, a newly generated register is used as the target of the subtraction instruction between the two registers, represented by the two areg nonterminals.

In order to have the difference between the two pointers in object count, instead of the byte count, this difference needs to be divided by the byte width of the pointers. The rule tries to insert an SRAI (shift right arithmetically). A single shift to the right represents a division by two. If the byte width was a power of 2, right shifts can thus be used. The amount of bits shifted is determined by the power of two computed earlier. SRAI is used, since the value in the register is stored in twos complement, SRAI respects this by shifting the value of the original most significant bit in the vacated upper bits. If however, the byte width was not a power of two, another register has to be generated. Into this register, the byte width integer value is then loaded with a MOV. Now at run time the difference in bytes, residing in the first register, is divided by the byte width, residing in the second one with a DIV instruction. This division necessitates the efforts to use a shift whenever

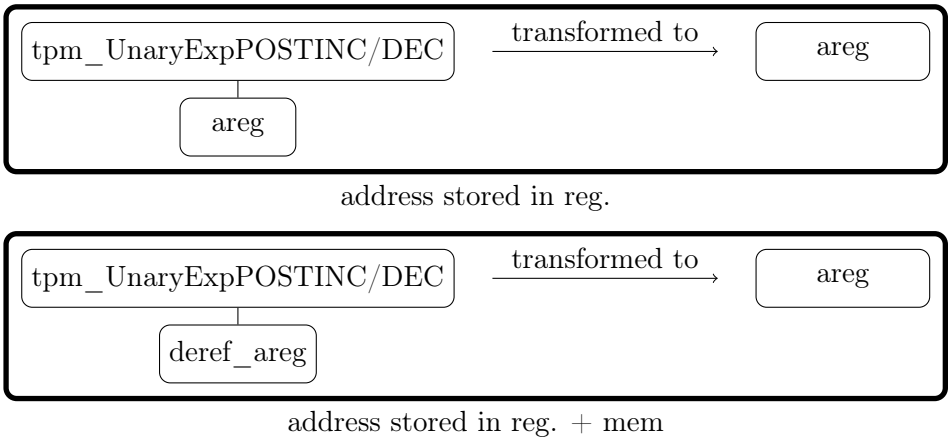

Figure 3.10: Postfix increments and decrements of pointers.

possible, as divisions are computationally harder than simple shifts. (Note that many data types have a byte width that is a power of two). The register holding the value of the division, can then finally be returned.

The costs for this rule first add the cost of a subtraction to the cost of the sub-tree, as there always will be a SUB instruction generated by this rule. Then depending on whether, the byte width is a power of two, costs for the SRAI instruction or the costs for the MOV and the DIV instructions are added.

3.6.5 Postfix and Prefix Increments and Decrements

Figure 3.10 shows the rules responsible for postfix increments and decrements of pointers. For the sake of simplicity, the tpm_UnaryExpPOSTINC and tpm_UnaryExpPOSTDEC are shown in a single node. Because of this, four rules can be seen in figure 3.10. As discussed in 3.5, the postfix increments and decrements are supposed to apply their respective modification but return the original value, in this case as an areg.

areg: tpm_ UnaryExpPOSTINC(areg) has an action part similar to the addition rules, as this is adding a value of one, but returning the original value of the address.

```
1 areg: tpm_UnaryExpPOSTINC( areg )
2 {
3   $cost[0] =
4     $cost[2] + RV32I::OperationFormat::RR_1.getSize() +
5     RV32I::OperationFormat::RRC12_1.getSize();
6 }
7 =
8 {
9   RV32::DEBUG_RULE_ACTION( "areg: tpm_UnaryExpPOSTINC( areg )", $1 );
```

```
10
11   auto *t = RV32::getBaseType( $2->getExp()->getType() );
12   int off = RV32::computeSizeOf( t );
13
14   auto &p = $action[2]();
15
16   auto &r = RVINSTRUCTIONS.createReg();
17   RVINSTRUCTIONS.insertCMV( r, p, $1->getExp() );
18   RVINSTRUCTIONS.insertADDI( p, p, off, $1->getExp() );
19
20   return( r );
21 };
```

For this, the byte width of the base data type has to be determined, which is done in line 12. Subsequently, the address value is moved into a newly created register via a CMV instruction in line 17. CMV checks whether the 32 bit MOV instruction is needed, and if not, uses the smaller 16bit variant. Afterwards, in line 20 the byte width as an integer value is added to the original register via an ADDI instruction. To finish, the rule returns the register r, which has the original value residing inside. This fulfills the postdecrement behavior, by returning the original value but incrementing the pointer itself. The cost for this rule is simply the cost of the addition (line 5) and the cost of the move(line 4).

areg: tpm_UnaryExpPOSTINC(deref_areg) essentially works the same way but due to the deref_areg, which is represented by an lvalue object, first, the register behind that object needs to be retrieved with ***getResultReg***. Then, the value of that reg can be copied over to a newly created reg with CMV. Afterwards, the register of the lvalue can be incremented in the same way as above with ADDI. Before returning the created register with the original value, however, the new value of the result reg is written back to memory with the ***storeBack*** function being called on the lvalue object. The costs here are almost the same as above, except for the addition of createStoreCost, to also take the write-back to memory into account.

The decrementation rules ***areg: tpm_UnaryExpPOSTDEC(areg)*** and ***areg: tpm_UnaryExpPOSTDEC(deref_areg)*** work much the same as their increment counterparts, but in a similar vein to the minus expressions earlier in this chapter, they simply take the negative of the offset. Other than this small modification they work the same way in the action part, making sure to return the original value and incrementing the pointer itself. Their cost parts thus also do not differ at all from the increment rules as the same instructions are used.

The rules representing prefix incrementation and decrementation of pointers can be seen in figure 3.11. There are four rules shown, due to the incrementation and decrementation terminals, which are merged into a node. As discussed in 3.5, for the preincrement rules, the action part is a bit easier, as

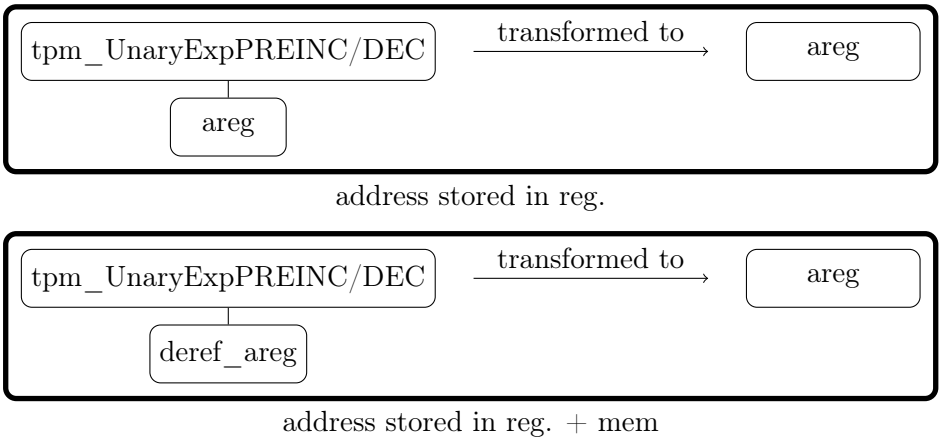

Figure 3.11: Prefix increments and decrements of pointers.

the pointer has to be incremented before returning its value.

```
1  areg: tpm_UnaryExpPREINC( areg )
2  {
3    $cost[0] =
4      $cost[2] + RV32I::OperationFormat::RRC12_1.getSize();
5  }
6  =
7  {
8    RV32::DEBUG_RULE_ACTION( "areg: tpm_UnaryExpPREINC( areg )", $1 );
9
10   auto *t = RV32::getBaseType( $2->getExp()->getType() );
11   int off = RV32::computeSizeOf( t );
12
13   auto &p = $action[2]();
14
15   RVINSTRUCTIONS.insertADDI( p, p, off, $1->getExp() );
16
17   return( p );
18 };
```

Thus in ***areg: tpm_UnaryExpPREINC(areg)***, the first step is the calculation of the byte width of the base data type of the pointer to be incremented (line 11). This offset is added via an ADDI instruction onto the register behind the areg nonterminal child node, in line 15. Since this is now also the value to be returned, nothing is to be done except the return of the modified register. The cost accounts for the ADDI instruction in line 4.

The rule ***areg: tpm_UnaryExpPREINC(deref_areg)***, matching against a deref_areg nonterminal, is more complex as additionally the register of the lvalue object needs to be retrieved. Also after the increment has been added to the register, this change needs to be written back to memory in the same way the ***areg: tpm_UnaryExpPOSTINC(deref_areg)*** rule did before, with ***storeBack***. This of course is also reflected in the costs with the additional createStoreCost added.

Figure 3.12: Assignment of pointers

The two predecrement rules ***areg: tpm_UnaryExpPREDEC(areg)*** and ***areg: tpm_UnaryExpPREDEC(deref_areg)*** again are the same as their increment counterparts, with the sole difference being the negative offset:

```
1 int off = -RV32::computeSizeOf( t );
```

This results in the wanted behavior as both rules first decrement their pointer by the byte width of their base data type and then return this decremented value.

3.6.6 Pointer-Assignment

In figure 3.12, the single rule, handling pointer assignments can be seen. It matches an areg on the right-hand side of the assignment. For the left-hand side, the ca_lhs_areg nonterminal, which was introduced in 3.4.2, is matched. The whole assignment returns a ca_result_areg.

ca_result_areg: tpm_AssignExpASSIGN(ca_lhs_areg, areg)

```
1  ca_result_areg: tpm_AssignExpASSIGN( ca_lhs_areg, areg )
2  {
3    if ( RV32::isArrayType( *$2->getExp() ) &&
4         dynamic_cast<IR_StringConstExp *>( $3->getExp() ) &&
5         !RV32::isFunctionArgument( *$2->getExp() ) )
6      $cost[0] = COST_INFINITY;
7    else
8      $cost[0] = $cost[2] + $cost[3] + RV32I::OperationFormat::RR_1.getSize();
9  }
10 =
11 {
12   RV32::DEBUG_RULE_ACTION(
13     "ca_result_areg: tpm_AssignExpASSIGN( ca_lhs_areg, areg )", $1 );
14
15   // Evaluate RHS first to account for side effects.
16   auto &p = $action[3]();
17   auto wbInfo = $action[2]( false );
18
19   RVINSTRUCTIONS.insertCMV(
20     dynamic_cast<RV_RegV &>( wbInfo.mTmpReg.get() ), p, $1->getExp() );
21
22   return( wbInfo );
23 };
```

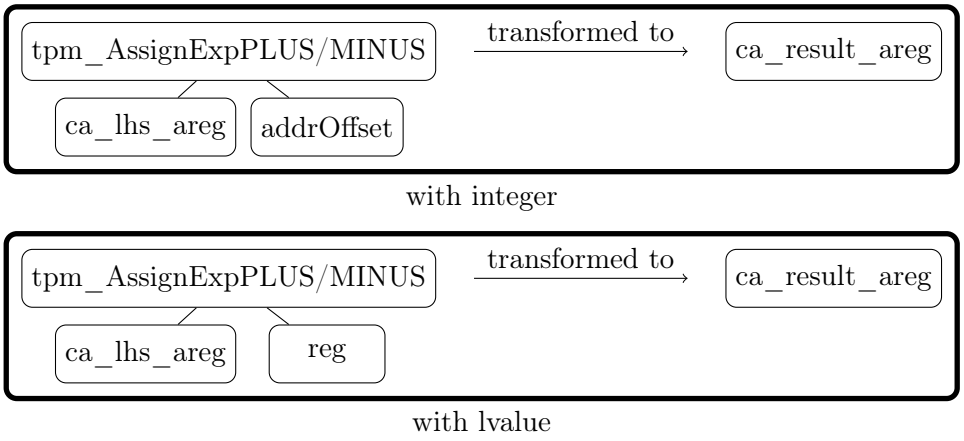

Figure 3.13: Assignment of pointers plus arithmetics

Lines 16 and 17 load the two nonterminal child nodes into their respective objects, i.e., the areg into a register object and the ca_lhs_areg into an RV32_WriteBackInfo object. Then, with a CMV instruction, the value of the areg nonterminal register is moved to the intermediate result register of the WriteBackInfo object(line 20). This WriteBackInfo object can be returned as ca_result_areg and also uses this as its representative.

ca_result_areg: tpm_AssignExpPLUS(ca_lhs_areg, addrOffset) is the first of four rules seen in figure 3.13. The figure again is simplified with the combination of AssginExpPlus and AssginExpMinus rules into one node. The AssginExpPlus rules handle the assignment-addition to a pointer. These rules work on C code like:

```
p += 1;
```

The end of section 2.5, explained that the expected outcome of this C code line is the incrementation of p. Effectively, these rules are a combination of an addition rule and an assign rule. As such, the two AssginExpPlus rules, are made for either the case that just an integer is used as an offset to be added or an actual register.

ca_result_areg: tpm_AssignExpPLUS(ca_lhs_areg, addrOffset) being the one for the integer case.

```
1 ca_result_areg: tpm_AssignExpPLUS( ca_lhs_areg, addrOffset )
2 {
3   $cost[0] = $cost[2] + $cost[3] + RV32I::OperationFormat::RRR_1.getSize() +
4   RV32I::OperationFormat::RRC12_1.getSize();
5 }
6 =
7 {
8   RV32::DEBUG_RULE_ACTION(
9     "ca_result_areg: tpm_AssignExpPLUS( ca_lhs_areg, addrOffset )", $1 );
```

```
10
11   auto wbInfo = $action[2]( true );
12   auto *t = RV32::getBaseType( $2->getExp()->getType() );
13   int off =
14     $action[3]().getIntValue() * RV32::computeSizeOf( t );
15
16   // done as MOVConstant + ADD since addi does not handle large numbers well
17   auto &temp = RVINSTRUCTIONS.createReg();
18   RVINSTRUCTIONS.insertMOVConstant( temp, off, $1->getExp() );
19
20   auto &r = dynamic_cast<RV_RegV &>( wbInfo.mTmpReg.get() );
21   RVINSTRUCTIONS.insertADD( r, r, temp, $1->getExp() );
22
23   return( wbInfo );
24 };
```

It starts, by calculating the offset as the multiplication of the integer value itself and the byte width of the base datatype of the pointer in line 14. Once the offset is calculated, this value can simply be added to the register holding the pointer itself.

Originally just an ADDI was used and the register was retrieved by again getting the register of the writeBackInfo object behind the ca_lhs_areg nonterminal. While in tests this worked for small numbers, large ones, specifically numbers exceeding 12 bits, ended up producing wrong outputs. The ADDI instruction implemented in the InstructionFactory does have the functionality to protect against overflow from integers larger than 12 bits.

```
1   // Generate LUI instruction.
2   insertLUI( xa, p.first, exp, type );
3
4   // Generate ADDI instruction.
5   if ( p.second != 0 )
6   insertADDI( xa, xa, p.second, exp, type );
7
8   insertADD( xa, xa, xb, exp, type );
```

For this, the ***insertADDI*** function checks whether the provided immediate is larger than 12 bits, if that is the case, the function calculates the value of the upper 20 bits and loads them with an LUI into the target register(line 2). The remaining 12 bits are then added with an ADDI in line 6. To finish up the addition, an ADD adds the value of the input register in line 8. This functionality seems to still have some imperfections, as the tests reveal wrong values residing in xa,

To circumvent this, the rule first uses an insertMOVConstant, in line 18 of the rules code, to load the offset into a temporary register and then uses an ADD instruction to add said offset to the register of the ca_result_areg nonterminal.

This rule is very similar to the ***areg: tpm_BinaryExpPLUS(areg, addrOffset)*** rule from earlier parts of this section. The important difference is, that here the register holding the pointer itself is modified, while in the

addition rule a new register is created to hold the resulting value of the addition, leaving the pointer register untouched. The costs of this rule reflect the costs of the movConstant and the ADD instruction added.

ca_result_areg: tpm_AssignExpPLUS(ca_lhs_areg, reg), the second assignment-addition rule, covers the case that a register is used to add to the pointer. First, the byte width of the base data type needs to be calculated. Then two new registers tmpReg1 and tmpReg2 are created. Into tmpReg1 the integer value of the reg nonterminal is moved via a MOV instruction. Afterward, the byte width is put into tmpReg2 with an insertMOVConstant. Now, tmpReg1 and tmpReg2 need to be multiplied with each other at run time via a MUL instruction. Finally, the result of that multiplication can be added onto the mTmpReg of the ca_lhs_areg nonterminal, which then can be returned. The costs are calculated by adding the sub-tree costs and the costs for the 4 used instructions.

Assign-subtraction rules ***ca_result_areg: tpm_AssignExpMINUS(ca_lhs_areg, addrOffset)*** and ***ca_result_areg: tpm_AssignExpMINUS(ca_lhsareg, reg)*** work much the same as their addition counterparts, with the first one making use of the negative offset. The second one has to use a SUB instruction instead of the final ADD instruction, as the offset is computed at runtime here. The costs also match the addition counterparts perfectly.

3.6.7 Related Rules

More rules were implemented, which do not match a terminal directly, or which do match a terminal that was not part of the terminals directly associated with pointers. These will not be explained as thoroughly as the rules above, however, a short description will be given for each.

- ***addrOffset: tpm_IntConstExp*** transforms an integer into an addrOffset nonterminal for further use.
- ***addrOffset: tpm_UnaryExpMINUS(addrOffset)*** transforms an addrOffset into its negative.
- ***areg: ca_result_areg*** transforms the result areg of an assignment back to its original form.
- ***areg: deref_areg*** transforms a register with a memory location into one without a memory location associated.

- ***areg: explicit_ castable_ reg***,
  areg: implicit_ castable_ reg
  cast a register designated to be cast into an areg.

- ***areg: modified_ areg*** this is a rule which actually is part of the pointer ruleset itself. Its job is to apply an outstanding modification of an areg. For this, it makes use of the underlying RV32_AddressModification object and its applyModification function.

- ***areg: tpm_ BinaryExpCOMMA(areg, areg)***,
  areg: tpm_ BinaryExpCOMMA(reg, areg)
  These rules do nothing by themselves but call both child nodes and return the one on the right subtree.

- ***areg: tpm_ CallExp(called_ function, arg)*** this rule handles function calls.

- ***areg: tpm_ ImplicitCast(constAddress)***,
  areg: tpm_ UnaryExpCAST(constAddress)
  these rules load a constant address into a register.

- ***arg: tpm_ CallExpARG(areg, arg)*** a rule for pointer in function calls.

- ***ca_ lhs_ areg: areg*** transforms an areg into the left side of an assignment.

- ***ca_ lhs_ areg: deref_ areg*** transforms a deref_areg into the left side of an assignment.

- ***constAddress: tpm_ IntConstExp*** transforms an integer into an address.

- ***explicit_ castable_ reg: tpm_ UnaryExpCAST(areg)***,
  explicit_ castable_ reg: tpm_ UnaryExpCAST(reg),
  implicit_ castable_ reg: tpm_ ImplicitCast(areg),
  implicit_ castable_ reg: tpm_ ImplicitCast(reg)
  rules dedicated to designate their register nonterminal to be cast.

- ***nrel: tpm_ BinaryExpEQ(areg, areg)***,
  nrel: tpm_ BinaryExpNEQ(areg, areg)
  rules checking for equal == or not equal != relations between addressees.

- ***reg: explicit_castable_reg***,
  reg: implicit_castable_reg
  rules which cast a register designated to be cast into a reg.

- ***reg: tpm_BinaryExpCOMMA(areg, reg)***,
  reg: tpm_BinaryExpCOMMA(reg, reg) these rules do nothing by themselves but call both child nodes and return the one on the right subtree.

- ***reg: tpm_BinaryExpGEQ(areg, areg)*** checking for >= relation

- ***reg: tpm_BinaryExpGT(areg, areg)*** checking for > relation

- ***reg: tpm_BinaryExpLEQ(areg, areg)*** checking <= relation

- ***reg: tpm_BinaryExpLT(areg, areg)*** checking for < relation

- ***reg: tpm_UnaryExpLOGNOT(areg)*** performs a logical negation on the areg register.

- ***reg: tpm_UnaryExpPREDEC(deref_reg)***,
  reg: tpm_UnaryExpPREINC(deref_reg) increments or decrements a register with an associated memory address.

- ***stmt: tpm_ExpStmt(areg)*** produces the last nonterminal and subsequently invokes all subsequent rules

- ***stmt: tpm_ReturnStmt(areg)*** represents the return with an pointer.

- ***zero_op_areg: tpm_IndexExp(areg, addrOffset)*** start of a &* operator succession.

Chapter 4

Experimental Results

The implemented rules and infrastructure around them needed to be tested properly for functionality and correct output. This was already a big part of the early work for this thesis as testing functionality was a good way of gauging the state of the RISC-V implementation. As such, there were two types of tests performed. The first was the manual tests which will be described in section 4.1, and the second was testing with the automated testbench. These tests are described in section 4.2 and play a large role in the experimental results. Unless otherwise noted, the information in this chapter stems from the "WCC – The WCET-Aware C Compiler Getting Started and Documentation" document [18].

4.1 Manual Tests

The manual tests are called such because here some form of C program was written and then manually compiled with the WCC. After the compilation, a manual inspection of the rules and the assembly code generated, was taken. This approach turned out to be beneficial as a testing methodology as the scope of the test program was very flexible. It it allows to quickly switch between a small two-line code or a few hundred lines. The activity of the codeselector could be monitored with the debug output of the compiler turned on. Furthermore, with the test programs, allowed easily switch between the RISC-V architecture and the TriCore architecture to validate rule routes being used by the RISC-V implementations. This feasible since the TriCore enjoys an already complete implementation. Additionaly, through the use of flags, while calling the WCC, it was possible possible to examine the assembly output during different stages or with different levels of optimizations applied.

To perform these tests, first off, the place where the test programs reside was usually in the build directory of the WCC and then under WCC/wcc. The WCC was called with ./wcc. The flags most commonly used were:

- mrudolv, to use the RISC-V implementation, leaving this out gives the output of the TriCore implementation, which is used by default.
- S to stop the compilation process after compiling so that the program is neither assembled nor linked. In certain cases it also is helpful here to use Sv, as the registers are not allocated with this flag, leaving the virtual registers in the assembly code
- vN, this flag sets the verbosity level to the specified number (N=0-9), which in turn determines the amount of debug output given. For the tests, a verbosity level of 2 was found to be the most fitting.
- ON, this flag is used to set the optimization level (N=0-3). Optimization is usually unwanted, so the source code to be tested usually is tested in its original form (N=0). The testbench, which will introduced in the next section 4.2 tests for every optimization level but sometimes, not all of them work. In these cases, this flag can be set to the defunct optimization level.
- filename, file to be compiled

With this setup, not only the rules taken by the tree pattern matcher can be validated, but the produced assembly code can also be checked for correct behavior.

A small C program, where an integer symbol with a value of 10 is created and then, a pointer p is assigned to i's address, was already partially introduced in section 2

```
int main()
{
  int i = 10;
  int *p = &i;
}
```

This program can be compiled with the WCC by executing:

```
./wcc -mrudolv -S -v2 -O0 test.c
```

During the compilation, in the code selector phase following rules are taken:

```
1  stmt: tpm_ExpStmt( reg )
2    i=10;
3  reg: ca_result_reg
4    [ i=10 ]
5  ca_result_reg: tpm_AssignExpASSIGN( ca_lhs_reg, reg )
6    [ i=10 ]
7  reg: tpm_IntConstExp
8    [ 10 ]
9  ca_lhs_reg: deref_reg
10   [ i ]
11 deref_reg: tpm_SymbolExp
12   [ i ]
13 ------------------------------------------------------------
14 stmt: tpm_ExpStmt( areg )
15   p=&i;
16 areg: ca_result_areg
17   [ p=&i ]
18 ca_result_areg: tpm_AssignExpASSIGN( ca_lhs_areg, areg )
19   [ p=&i ]
20 areg: tpm_UnaryExpADDR( deref_reg )
21   [ &i ]
22 deref_reg: tpm_SymbolExp
23   [ i ]
24 ca_lhs_areg: areg
25   [ p ]
26 areg: tpm_SymbolExp
27   [ p ]
28 ------------------------------------------------------------
29 stmt: tpm_ReturnStmtVOID
30   return;
```

The listing above, is the simplified debug output of the compilation process. The used rules are separated into three sections, the first one and the second one representing the first and second line of the C code, while the last section symbolizes the return of the main function.

At line 7 and 12, there are two expressions, first, i represented by the tpm_SymbolExp terminal. Second, the integer 10 is represented by the tpm_IntConstExp terminal. The i terminal gets converted to a deref_reg nonterminal(line 11), which then gets converted into a ca_lhs_reg nonterminal(line 9). The 10 terminal gets converted into a reg nonterminal(line 7). With ca_lhs_reg and reg, now the assignment of 10 to i can be performed(line 5), followed by a conversion from the resulting ca_result_reg to a normal reg(line 3). With the reg nonterminal, the entire statement finds its end with the stmt: tpm_ExpStmt(reg) rule(line 1), which produces the stmt nonterminal representing the entire tree.

At the bottom of the second section two expressions are present. This time, they both are symbol expressions, one for i (line 23)and one for p(line 27). In a similar vein to the first section, the tpm_SymbolExp terminal representing p gets converted into an areg and then into a ca_lhs_areg. The tpm_SymbolExp terminal representing i first gets translated into a deref_reg. With this nonterminal as the child node

of the tpm_UnaryExpADDR, used to represent the address operator applied to i, the rule areg: tpm_UnaryExpADDR(deref_reg) can be applied which extracts i's address. This address is then assigned to p in ca_result_areg: tpm_AssignExpASSIGN(ca_lhs_areg, areg). After converting ca_result_areg into an areg, the whole section is once again completed with stmt: tpm_ExpStmt(areg).

The assembly code resulting from these rules can be seen in listing below.

```
1   main:
2       addi            x2, x2, -64
3       sw              x1, 60(x2)
4       sw              x8, 56(x2)
5       sw              x9, 52(x2)
6       sw              x18, 48(x2)
7       sw              x19, 44(x2)
8       sw              x20, 40(x2)
9       sw              x21, 36(x2)
10      sw              x22, 32(x2)
11      sw              x23, 28(x2)
12      sw              x24, 24(x2)
13      sw              x25, 20(x2)
14      sw              x26, 16(x2)
15      sw              x27, 12(x2)
16      addi            x8, x2, 56
17      addi            x5, x0, 10
18      sw              x5, -56(x8)
19      addi            x5, x8, -56
20      c.mv            x5, x8
21      lw              x27, -44(x5)
22      lw              x26, -40(x5)
23      lw              x25, -36(x5)
24      lw              x24, -32(x5)
25      lw              x23, -28(x5)
26      lw              x22, -24(x5)
27      lw              x21, -20(x5)
28      lw              x20, -16(x5)
29      lw              x19, -12(x5)
30      lw              x18, -8(x5)
31      lw              x9, -4(x5)
32      lw              x8, 0(x5)
33      lw              x1, 4(x5)
34      addi            x2, x5, 8
35      jalr            x0, x1, 0
```

Lines 1-16 and 20-35 deal with saving and restoring of registers for a function call and return. The most relevant lines for this thesis are 17-19. In line 17, the value 10 is added to the register x0 and the result is saved in register x5. Here, the creation of the symbol i can be observed, as the value 10 is effectively loaded into register 5 due to x0 being hardwired to the value 0. Since i is an actual lvalue and the memory location is important since it is retrieved later on, i also has a designated memory location. In line 18, the value of i is saved to its memory location, which is the value saved in x8 with an offset of -56. Line 19 marks the creation of the pointer symbol p. Here, the memory address of i is written into the register x5. To create the

address, the value of x8 is added to the offset -56 and the result is then saved in x5. These instructions now effectively create the following process: First the integer symbol i is created, and its value is saved to memory. Second, the pointer symbol p is created and pointed at the address of i. This represents the C code.

Adding a third line to the program which dereferences the pointer and saves the resulting value in an integer variable j

```
int main()
{
  int i = 10;
  int *p = &i;
  int j = *p;
}
```

produces the following assembly lines (only the lines of interest are shown):

```
        addi            x5, x0, 10
        sw              x5, -56(x8)
        addi            x5, x8, -56
        lw              x5, 0(x5)
```

The first three lines of interest did not change, however, the fourth line now consists of an LW instruction. This instruction loads a value from the memory address saved in x5, i.e., the pointer, and places this value into register x5, subsequently creating the symbol j.

It might be confusing that the symbols seem to overwrite each other. However, since each time an old symbol is overwritten, it actually is not important for the further computation of the program anymore and thus it is not of importance anymore to store that symbol.

```
int main()
{
  int i = 10;
  int *p = &i;
  p += 1;
}
```

Replacing the third line of the program by a line which increments the pointer p by one produces the following assembly lines (only the lines of interest are shown):

```
addi            x5, x0, 10
sw              x5, -56(x8)
addi            x5, x8, -56
addi            x6, x0, 4
add             x5, x5, x6
```

In addition to the three lines from the first test, there now is an ADDI instruction to save the byte-width of an integer data type in x6. This byte-width (4) is then added to x5 to increment the pointer. The assembly output matches the C source in functionality.

4.2 Testbench

The testbench is made for testing a wide range of functionality in an automated manner. For this, it is made up of multiple thousands of tests consisting of ANSI C source code. The testbench is divided up into a real-world part and a positive part. For the results of this thesis, the positive part is of special interest. The positive part of the testbench is further divided up into tests for specific data types, like int, char, float and so on. As the integer data type is the first one to be implemented since, and for the RISC-V architecture the goal was to successfully compile the int pointer tests, the int tests are the most important ones out of the specific data types.

The testbench can either call single tests or a group of tests. To call a single test, a command similar to

```
./intpointer01.sh --verbose 3
```

can be executed. Here, the first int pointer test is called with a verbosity level of three. In the best case, one can now observe the testbench compiling the test program with four different optimization levels. The four generated assembly files are then executed and their output is checked against a version of the test that was compiled with the GCC compiler. If all four versions of the test compile successfully and give the right output, a success message is generated. In case of a malfunction the testbench outputs an error message into the terminal. As described, the testbench is made for large-scale testing of the WCC for which it would be tedious to call each test manually. Thus, one can execute all tests within a sub-folder with

```
make -j 20 check
```

The -j 20 flag marks the use of 20 threads. After going through all tests, a general summary is printed in the terminal of how many tests passed, how

many tests failed and how many were skipped. Before that, for each test, there is an output on whether it passed or not.

In the testbench, there are 44 pointer tests for the integer type. They are numbered with 1-48 where 10, 20... are left out. Each pointer test usually checks a certain part of pointer functionality with many different combinations of operators and values. In doing so, the tests and what they check for, are divided by the last digit in their number. The last digits can be summarized as follows:

- 1 Check taking and dereferencing a pointer locally.
- 2 Check taking and dereferencing a pointer globally.
- 3 Check taking and dereferencing a static pointer.
- 4 Check passing of pointers as function parameters/return values. Check assignments to a pointer. Check sizeof computations for pointers. Check logical negation of pointers.
- 5 Check equality/inequality of pointers, for local global, static.
- 6 Check relational operators of pointers, for local global, static.
- 7 Check pointer arithmetic local.
- 8 Check pointer arithmetic global.
- 9 Check pointer arithmetic static.

This results in intpointer01.c starting out by testing for taking and dereferencing a pointer locally, like int *p. While intpointer31.c starts by testing for taking and dereferencing a pointer to a local struct, like struct globStruct *p. At the start of this thesis, none of the 44 pointer tests could be successfully compiled. Afterwards, 35 of the tests lead to success. The other 9 are the intpointer31-39 tests. These rely strongly on composed types such as structs. Since there is no composed type ruleset yet and especially no composed type infrastructure for the RISC-V implementation, these tests could not be completed during this work.

In addition to the 35 pointer tests now being compiled correctly, some tests outside of the pointers also can be successfully compiled now. A few of these tests required rules which are now implemented (e.g., the type casting ruleset) and some were being held off by errors that were fixed, within the rulesets infrastructure. Before this thesis, 213 tests passed in the int subfolder of the positive testbench. After the implementation of the ruleset

from this work, 279 tests pass successfully. This includes the 35 passing tests of the int pointer type. Thus, 31 other tests now additionally pass as well. Similar data types to int also now have succeeding pointer tests. The pointer tests in the long testbench also all work, except the 31-39 which deal with composed types, the same goes for the uint tests. For the void testbench, all but one pointer test succeed and for the ulong, half of the pointer tests succeed (31-39 plus last digits of 1, 2, 3 still fail). An overview of the tests passing in the positive testbench can be seen in table 4.1. Note that for the state of rules successfully compiled, the thesis [5] is taken as the reference point. The table shows an increase in passed tests in many sub-testbenches,

Datatype	**Amount of tests**	**Passed before**	**Passed after**
bool	327	5	5
cfes 1	83	56	82
cfes 2	20	19	19
cfes 3	8	0	0
char	409	0	137
double	368	0	0
float	369	0	0
int	463	213	279
long	331	1	167
longdouble	364	1	1
longlong	444	0	1
minimal	327	18	63
misc	25	0	3
short	327	1	109
string	11	0	4
uchar	398	0	167
uint	302	2	147
ulong	302	2	133
ulonglong	435	0	1
ushort	317	0	91
void	20	0	18

Table 4.1: positive testbench results

however bool, double, float and longlong, plus their related data types still have minuscule advancements in passed tests. This is to be expected, since these data types need special rules to be handled correctly. For example, in

the case of float, the rv32IMC implementation of RISC-V might even require an expansion, upon the F extension [16, p. 45].

Chapter 5

Conclusion and Outlook

5.1 Conclusion

The goal of this thesis was to implement the address operator & and the indirection operator * within the RISC-V code selector of the WCC compiler. Furthermore, pointer-arithmetic like $+=$, $-=$, $++$, $--$ and pointer as function arguments as well as returning functions were set out to be operational as well. With respect to chapter 4, these goals have been achieved to the point where the int pointer tests, except the ones relying on a composed type ruleset, now pass compilation. This indicates that both operators, pointer-arithmetic and pointer as function arguments/returns are functional for the integer datatype.

To conclude the work done during this thesis, two new classes were introduced. Which were the AddressWithOffset class, which has been completed and the Cast class, which still lacks functionality. Both classes however support the needed functionality for pointer-related C code and produce correct assembly code for the tests applied in chapter 4. Important to mention that the cast class was only tested with the int tests, with attention to the int pointer tests. This insinuates that other data types may still produce errors or are not yet included in the case distinction inside of doCasting (for the char type, a potential error was found already).

The implemented ruleset has been brought to a state where the goals were met. Furthermore as already noted in section 3.6, the created rules do not all directly relate to the pointer ruleset. Some, like the cast rules, were implemented in order to expand functionality needed for tests from the testbench.

5.2 Outlook

The work on the RISC-V architecture within the WCC has to be continued. This is indicated by more than half of the positive testbench tests not being successfully compiled. Thus, there are several more steps to be taken towards a full implementation.

One of the immediate steps might be the expansion of the cast class. All cases of possible source and destination data types need to be implemented, as well as the addition of more functions, especially, cost functions, which were not included yet. This should also be accompanied by an expansion of the cast ruleset.

Another next step would be the addition of the composed type ruleset and auxiliary functions. This would have an immediate effect on this pointer ruleset, completing the last 9 tests for the integer data type and possibly for some related data types as well.

Beyond that, other data types like bool, char, and so on, also require implementation.

In the future, when all of the above steps have been done, further implementations of other RISC-V instruction-set expansions might be conceivable. It may even be viable to turn support for certain extensions on and of, on the go.

List of Figures

List of Tables

Bibliography

[1] Peter Marwedel. *Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer, 4 edition, 2021.

[2] Jonas Oltmanns. Erweiterung einer low-level compiler-zwischendarstellung um die risc-v befehlssatz-architektur. Bachelor's thesis, Hamburg University of Technology, 2021.

[3] Maurice Hoffmann. Entwurf und evaluation einer bit-genauen daten- und wertfluss-analyse für den risc-v imc-befehlssatz. Bachelor's thesis, Hamburg University of Technology, 2022.

[4] Ben Bahe, Christian Sühl, Nia Genadieva, Rasmus Mecklenburg, Reine Ngamo, Ruben Kuhlmann, and Simon Kopischke. Erweiterung des wcc-compilers im rahmen des riscy-praktikums. 2022.

[5] Christian Sühl. Stack frame allocation inside a compiler for the risc-v processor architecture. Bachelor's thesis, Hamburg University of Technology, 2022.

[6] Hans-Jürgen Schneider. *Compiler Aufbau und Arbeitsweise*. Walter de Gruyter, 1975.

[7] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real- Time Systems*. Springer, 2011.

[8] AbsInt Angewandte Informatik GmbH. Worst-case execution time analyzer ait, 2022.

[9] JaninaPlog. Developing a code selector's ruleset for the arm processor. Bachelor's thesis, Hamburg University of Technology, 2017.

[10] ICD – Informatik Centrum Dortmund. *ICD-C Compiler Framework Developer Manual*. 2022.

[11] Heiko Falk. Compilers for embedded systems, Summer Term 2022.

[12] Tobias Marschner. Designing and testing a code selector ruleset for jump-statements and pointer-expressions targeting arm processors as part of a wcet-aware compiler. Bachelor's thesis, Hamburg University of Technology, 2019.

[13] Guido Costa Souza de Ara´ujo. *CODE GENERATION ALGORITHMS FOR DIGITAL SIGNAL PROCESSORS*. PhD thesis, princeton university, 1997.

[14] Heiko Falk. Compilers for embedded systems, Summer Term 2022.

[15] Western Digital Editorial Team. Risc-v swerv core™, 2022.

[16] Andrew Waterman, Krste Asanovic, and SiFive Inc. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. CS Division, EECS Department, University of California, Berkeley, 2.2 edition, 2017.

[17] International Organization for Standardization. *Programming languages - C*. International Organization for Standardization, Vernier, Geneva, Switzerland, iso/iec 9899 second edition edition, 1999.

[18] Arno Luppold, Dominic Oehlert, Mikko Roth, Shashank Jadhav, and Heiko Falk. Wcc – the wcet-aware c compiler getting started and documentation. 2022.

[19] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[20] Heiko Falk. Compilers for embedded systems. 2022.