CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Formally Verified Compiler for Programs that Deal with Cached Address Translation

Master's thesis in Computer science and engineering

JOHAN ANDERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

A Formally Verified Compiler for Programs that Deal with Cached Address Translation

JOHAN ANDERSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

A Formally Verified Compiler for Programs that Deal with Cached Address Translation
JOHAN ANDERSSON

Supervisor: Hira Taqdees Syeda, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

A Formally Verified Compiler for Programs that Deal with Cached Address Translation

JOHAN ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Security in modern operating systems ultimately relies on the paging mechanism of the processor, which in turn relies upon the translation lookaside buffer (TLB). One method of showing the correctness of operating systems is formal verification, but previous work has left the correctness of the TLB as an assumption. Later work on the correctness of the TLB and high-level languages respectively leaves a gap.

In this report, we investigate how to develop and verify a compiler that preserves the TLB consistency of high-level languages, bridging the gap. We describe the considerations when implementing such a compiler, the roadblocks encountered, and the result. We end with stating the compiler correctness theorem and then discuss the state of the compiler and future work.

Keywords: Compiler, Translation lookaside buffer, Verification, Formal methods, Language.

Acknowledgements

We would like to thank Hira Taqdees Syeda, Magnus Myreen, and the other members of the Formal Methods division at Chalmers University of Technology for all assistance when writing this thesis.

Johan Andersson, Gothenburg, June 2020

Contents

1
Introduction

Today we expect our computer systems to run multiple programs concurrently in a secure manner. It is the responsibility of the operating system kernel (hereafter referred to as the kernel) to enable this by managing the hardware such that applications cannot interfere with each other's memory. The kernel is the low-level component of an operating system, managing the system resources such as the processor, memory, and I/O-devices.

The kernel uses virtual memory to give each process running on the system its own virtual address space, giving each process the illusion of being the only process running on the system.

The scheme deployed to implement virtual memory is called paging, which divides the physical memory into pages of a fixed size. Pages, which are identified by physical addresses, are mapped into virtual address spaces using data structures called page tables. The operating system manages these tables. The page tables are read by a memory management unit (MMU), which uses the page tables' content to translate mapping from virtual to physical addresses and perform security checks.

One problem that arises when using paging is that each memory access done by an instruction requires the MMU to do multiple lookups in the page tables, slowing down the system considerably. Therefore, processors use a mechanism called translation lookaside buffer (TLB), which caches results from the page tables. Hence, it is the TLB that ultimately provides process isolation. Since paging is the foundation of security when it comes to operating systems, ensuring it always works as expected is paramount.

Formal verification is one way to verify the correctness of the implementation of code that interfaces with the MMU. However, since the kernel interacts with the TLB using machine language, such verification must be performed using the machine language itself or on high-level code compiled using a verified compiler. Using such a compiler, we can verify that it does not introduce TLB inconsistencies in the resulting machine code.

In this thesis, we present a verified compiler for a language supporting statements for interacting with the TLB. We show the correctness of the compiler, and that the correctness of high-level code can carry over to the machine code outputted by the said compiler.

1.1 Research questions

We define our research questions for this project as follows: Is it possible to prove that a compiler preserves TLB consistency when compiling source code into machine code? Do a property proved in a high level program propagate down to the low level machine code?

1.2 Scope

This thesis aims to implement a compiler aware of the TLB and state the compiler correctness theorem. We limit ourselves to implementing a simple compiler without any optimization or other advanced compiler features. We do not intend to reinvent previous work done in the field. Instead, we choose to adapt our work to previous work for as much as possible, only adding to or changing the previous work when necessary. When it comes to proving the compiler correctness theorem, we realize that we probably will not be able to do this entirely within the given timeframe. Hence, we choose to show as much of the theorem as possible without considering a partial proof a failure.

1.3 Contribution

This thesis shows that developing a compiler, as defined in our research question, is feasible. We show such a compiler, together with the corresponding compiler correctness theorem. While leaving the correctness of some compiler functions as assumptions, we prove that the compiler preserves the TLB consistency for expressions and statements specified by the language.

1.4 Outline

The rest of this report have the following outline:

In chapter 2, we present previous work on the subject of this thesis, and the language and CPU concepts used in the project. We also introduce the syntax of Isabelle/HOL, the general theorem prover we use in this project.

In chapter 3, we present the general principles behind the functions and lemmas used by the compiler and the correctness theorem. We show why we chose these principles and how they relate to the background presented in chapter 2.

In chapter 4, we present the final compiler and correctness theorem in detail. We show the compiler implementation of each construct of the language and corresponding correctness proofs. In the cases where we have not been able to prove a construct, we show the assumptions we use.

In chapter 5, we discuss the tradeoffs made when developing the compiler and the proof, and which improvements that are possible. We also discuss the assumptions made and the reason for not proving them.

In chapter 6, we summarize our findings and present our conclusions.

2
Background

In this chapter, we give the background that the reader needs to know when reading this report. We start by describing the literature survey that we did when starting the project, and then we describe the architecture, concepts, and previous work that this report revolves around. We end the chapter by describing Isabelle/HOL, which is the theorem prover used in the previous work we build our project upon, and which we also use.

2.1 Literature survey

Klein et al. present seL4, the first formally verified kernel [1] [2] building upon the L4 family of microkernels. Gu et al. present CertiKOS [3]. Having a verified kernel provides a first step towards solving the problem with TLB inconsistency described in chapter 1. The authors of these projects show the feasibility of such kernels, but assume the correctness of the underlying hardware, including the TLB.

Using L3 [4] [5], a specification language for Instruction Set Architectures (ISA), Fox and Myreen model the ARM ISA [6], thereby making it possible to reason about programs compiled for ARM. In their work, the authors model the architecture down to the machine code level, making it possible to reason about the architecture's behavior down to the instruction cycle. While the authors leave the correctness of the TLB as an assumption, their model provides an important step towards proving TLB consistency. In this report, we will refer to this model as the Cambridge ISA.

Syeda and Klein present a formal model for reasoning about the MMU for the ARM architecture [7]. In their work, the authors use Isabelle/HOL [8] to model the ARM MMU, including TLB and page tables, and they prove the correctness of page table walks when using the ARM ISA [9]. The authors integrate their proof into the Cambridge ISA, establishing a ground truth for reasoning about low-level programs in the presence of a TLB.

Syeda and Klein further present a formal model and develop a logic for a high-level programming language for reasoning about an abstract MMU [10]. In their work, the authors use Isabelle/HOL to prove the correctness of page table walks when using a high-level language with support for TLB management primitives. The authors conclude that a gap exists between the MMU model and the high-level

model. The authors argue that further investigation is necessary to prove that their high-level model holds for a concrete system. In the discussion, the authors ask if it is possible to bridge the gap in the logic using a compiler aware of the TLB.

2.2 ARM

In this section, we describe the ARM Architecture [9], the architecture used in the previous work we build our project upon. The ARMv7 architecture is a reduced instruction set computing (RISC) architecture featuring multiple execution modes, and an MMU with a TLB. These features make it possible to execute a secure operating system kernel on the architecture. Besides having an MMU, ARM also has typical RISC features such as load/store instructions for simple addressing, fixed instruction length (32 bits), and a large (13) number of general-purpose registers. In contrast to many other architectures, ARM also features conditional execution of most instructions, making it possible to reduce the number of branches to a minimum.

2.2.1 Execution modes

The ARM architecture features multiple execution modes, of which we describe supervisor mode and user mode in particular. Programs executing in supervisor mode (typically an operating system kernel) have privileged access to the resources in the system and can, therefore, modify properties such as system registers and page tables. In contrast, the processor restricts programs running in user mode (typically all programs but the kernel), preventing them from accessing system resources available in supervisor mode. In this report, we will refer to supervisor mode as kernel mode. For a complete description of the execution modes available on ARM, see Table B1-1 in the RM Architecture Reference Manual [9].

2.2.2 MMU

The ARM Architecture features a memory management unit (MMU) that provides support for virtual memory, a method for the operating system (OS) to control access to the memory of processes. The OS gives each process its own private address space of virtual addresses, which it maps to physical addresses in the computer's memory using page tables stored in memory. One can think of the page tables as a dictionary, where the virtual addresses are the keys, and the physical addresses are the values.

When virtual memory is enabled, all addresses used by processes (including the OS) are virtual. When a process accesses some address, the MMU performs a so-called page table walk to find the corresponding physical address and perform security checks. The MMU does the page table walks by reading the page tables from memory and interpreting their content.

Doing page table walks implies that the MMU accesses memory itself, which is slow

compared to accessing resources internal to the processor. Therefore, the MMU features a translation lookaside buffer (TLB), a caching scheme for page table walks. While the TLB speeds memory access, it also introduces potential problems. It becomes possible for the operating system to introduce inconsistency between the page tables and the results from page table walks stored in the TLB, potentially enabling processes to access memory in a way not intended by the OS. To rectify this problem, the System Control coprocessor (which is the mechanism controlling the virtual memory on ARM) provides a mechanism the OS can use to flush (invalidate) data stored in the TLB, either partially (by address, process, or both) or entirely. Coprocessors are a way to extend the functionality of the ARM architecture, and the architecture implements access to many hardware-related features using them.

2.2.3 Registers

In this subsection, we describe the registers that we refer to throughout this report.

Firstly, ARM features thirteen general-purpose registers named `R0` to `R12`, also referred to as the core registers. With a general-purpose register, we mean registers that do not have any special function and are accessible in all execution modes.

In addition to the general-purpose registers, there is also `SP`, `LR`, and `PC`. These are the stack pointer, the link register, and the program pointer, respectively. We mention all of these for completeness while we, out of these registers, only reason about `PC` in this report. While not general-purpose, these registers are also accessible in all execution modes.

Another register that occurs throughout this report is the `CPSR`, which stands for Current Program Status Register. We show the fields of this register that we use in this report in table 2.1. For more information about this register, we refer to B1.3.3 in the ARM Architecture Reference Manual [9].

N	1	Negative condition flag
Z	1	Zero condition flag
C	1	Carry condition flag
V	1	Overflow condition flag
J	1	Jazelle mode enabled
E	1	Endianness
T	1	Thumb mode enabled
M	5	Mode

Table 2.1: Fields of CPSR that we use in this report. The first column is the field name, the second column is the size in bits, and the third column is the notation we use throughout this report. The first four fields, N to V, are the condition flags. When executing some arithmetic/logic computations, the processor updates these flags according to the result. The next three fields, J to T, are flags for enabling processor modes other than ARM. The last field, M, determines the execution mode of the processor (described in subsection 2.2.1). If the value of M equals 0x10, the processor runs in user mode. If it equals 0x13, the processor runs in kernel mode.

Finally, there is the TTBR0 register, which holds the page table root (the physical address from which the MMU starts page table walks) and the ASID, which is the current process identifier. This register is not a true ARM register since the System Control coprocessor handles it. For our purposes, however, it suffices to know that the TTBR0 is available using a privileged instruction.

In addition to the registers described in this subsection, there are further registers we do not mention. For a complete overview of the registers on ARM, we refer to the ARM Architecture Reference Manual [9].

2.3 Isabelle/HOL

In this section, we describe Isabelle/HOL, a general theorem prover with support for first-order logic and high-order logic (HOL), and used in the previous work that we use as a basis for this project. In this report, we will use the names Isabelle/HOL and Isabelle interchangeably.

Using Isabelle and other general theorem provers such as Coq [11], the user provides both a specification and an implementation. The user then proves that the implementation matches the specification using different proof techniques, which can be both automated and interactive. When doing interactive proofs, the user applies logic rules and lemmas, which can be both user-defined as imported from libraries. Using formal verification to prove the consistency between a specification and an implementation greatly increases the confidence in a system.

Since we use a lot of Isabelle notation in this report, we spend the rest of this section describing the syntax of Isabelle that we use in this report. For a complete overview of the syntax of Isabelle, we recommend consulting the documentation [8]

and the literature [12].

2.3.1 Data types

In this subsection, we show the data types provided by Isabelle and examples of how to use them. We also show to create user-defined types. We restrict ourselves to the types used in this report. For a complete description of the Isabelle type system, we refer to the Isabelle Documentation [8].

Table 2.2 shows the **base types** provided by Isabelle. These type are both used directly and to construct more complex data types.

`bool`	$\{True, False\}$	Booleans
`nat`	$\mathbb{N}$	Natural numbers
`int`	$\mathbb{Z}$	Integers

Table 2.2: Base types used in this report. The first column is the name used in Isabelle, the second column is the set of values the type can have, and the third column is the notation we use for the type throughout this report.

List types consist of some type `T` followed by the word `list`. There are several operators associated with list, and we show the ones we use in this report in table 2.3.

The first and second operators in table 2.3 are the start-of-list and end-of-list operators. Placing elements separated by comma between these, it is possible to create a new list. For example, `[1,2,3]` creates a new list containing the elements `1`, `2`, and `3`.

The third operator is the prepend element operator. This operator takes an element of some type `T` as its left-hand operand, a list of type `T` as its right-hand operand, and returns a new list where the left-hand element is the first element, followed by the elements in the right-hand list. For example, `1 # [2,3]` creates a new list containing the elements `1`, `2`, and `3`.

The fourth operator is the prepend list operator. This operator takes a list of some type `T` as its left-hand operand, a list of type `T` as its right-hand operand, and returns a new list where the elements in the left-hand list are the first elements, followed by the elements in the right-hand list. For example, `[1] @ [2,3]` creates a new list containing the elements `1`, `2`, and `3`.

`[`	start-of-list
`]`	end-of-list
`#`	prepend element
`@`	prepend list

Table 2.3: List operators used in this report. The first column is the operator and the second column is the notation we use.

Function types consist of some types separated by the type separator (shown in table 2.4), where the last type is the return type.

⇒	`=>`	type separator

Table 2.4: Function type symbols used in this report. The first column is the operator, the second column is the ASCII representation of the operator, and the third column is the notation we use.

```
"nat => nat => nat"
```

Listing 2.1: An example of a function type. The type represents some function that takes two arguments of type `nat`, and returns a value of type `nat`. For example, it is possible to represent addition of natural numbers using this function

It is possible to have **user-defined types** using the keyword `datatype`. When defining a new type, the user provides a name and possible values. In turn, the values can have properties associated with them.

```
datatype vehicle = bicycle |
                   motorcycle nat |
                   car nat
```

Listing 2.2: An example of a user-defined type as they appear throughout this report. The `vehicle` type can have three different values, with two having a property. The property can, for example, represent the minimum age for driving the vehicle, or anything else the user wants.

Finally, there are also **record types** in Isabelle. These types are data structures built from other types, and are associated with multiple operators. While we do not define any records of our own in this report, we still use operators related to records. We show these operators in table 2.5.

⦇	`(|`	start-of-replacement
⦈	`|)`	end-of-replacement
	:=	assignment

Table 2.5: Record operators used in this report. The first column is the operator, the second column is the ASCII representation of the operator, and the third column is the notation we use.

Listing 2.3 shows an example of how we use the operators in table 2.5. The first operator is the start-of-replacement operator, followed by some assignments, which are name-value pairs. The assignment operator separates the names and values, and a comma separates the pairs. The replacement ends with the end-of-replacement operator.

```
"person(|age := 50, height := 180|)"
```

Listing 2.3: An example of a record replacement as they appear throughout this report. These operators will return a new record which retains all fields from `person`, except for `age` and `height`, which it sets to the values given.

2.3.2 Formulas

Isabelle supports formulas, which are expressions with return type `bool`. When working with formulas, it is possible to use the usual operators from first-order logic. In table 2.6, we show the operators we use in this report.

∀	`ALL`	for all
∃	`EX`	there exists
∧	`&`	and
⟶	`->`	implies
∨	`|`	or
¬	`~`	not

Table 2.6: Logical symbols used in this report. The first column is the operator,the second column is the ASCII representation of the operator, and the third columnis the notation we use

2.3.3 Definitions

Listing 2.4 shows definitions as they appear in this report. We use the Isabelle keyword `definition`, followed by the name and two colons. A function type (described in subsection 2.3.1) follows. After the type, we find the `where` keyword, followed by a string containing the body of the definition. The body consists of the definition name, the variable names, the separator `==`, and then an expression.

```
definition
  xor  :: "bool => bool => bool"
where
  "xor a b == (a | b) & ~(a & b)"
```

Listing 2.4: An example of a definition as they appear throughout this report. The definition name is `xor`, it takes two arguments of type `bool`, and it returns a value of type `bool`. The body states that an expression `xor a b` is equivalent to `(a | b) & ~(a & b)`.

2.3.4 Functions

Listing 2.5 shows functions as they appear in this report. Except for starting with the keyword `fun` and having the ability to consist of more than one body, functions

have almost the same syntax as definitions. The first body whose guard is true will execute when calling the function.

```
fun
  factorial  :: "nat => nat"
where
  "factorial 0 == 1" |
  "factorial n == (factorial (n-1)) * n"
```

Listing 2.5: An example of a function as they appear throughout this report. The function name is `factorial`, it takes one argument `n` of type `nat`, and it returns a value of type `nat`. There are two cases; if `n` is zero, then the function returns one. Otherwise, the function executes a recursive call with `n` decremented as argument, and returns `n!`.

2.3.5 Proofs

Listing 2.6 shows how we represent proofs in this report. We use the keyword `lemma`, followed by the name and colon. After the name, there follows a string containing our assumptions, followed by our conclusion. Both assumptions and the conclusion are formulas (i.e., they are of type `bool`). The assumptions are separated by a semicolon and located between the brackets we show in table 2.7. The brackets are optional in case there is only one assumption. Finally, the conclusion follows after the implies symbol shown in table 2.7.

When using Isabelle, a proof body follows directly after the string containing the assumptions and the conclusion. In this proof body, Isabelle expects the user to apply the proof methods necessary for proving all subgoals of the proof. While we, for readability reasons, omit the proof body when stating proofs in this report, it is essential to understand that it is there.

```
lemma conjunct:
  "[|A; B|] ==> A & B"
```

Listing 2.6: An example of a proof as they appear throughout this report. The proof name is `conjunct`, it assumes `A` and `B`, and it concludes `A & B`. In other words, the proof states that if `A` and `B` holds, then `A & B` holds.

⟦	`[|`	start of assumptions bracket
⟧	`|]`	end of assumptions bracket
⟹	`==>`	implies

Table 2.7: Symbols occurring in proofs and assumptions. The first column is the operator, the second column is the ASCII representation of the operator, and the third column is the notation we use.

2.3.6 Assumptions

Listing 2.7 shows how we represent assumptions. They have the same syntax the proofs described in subsection 2.3.5, except that instead of having a proof body, they end with the keyword `sorry`.

When Isabelle parses the keyword `sorry`, it accepts the lemma as a proof without the user having to provide the proof body. Hence, lemmas having `sorry` instead of a proof body are not proofs but assumptions.

In subsection 2.3.5, we state that we omit the proof body when stating proofs in this report. However, to distinguish between proofs and assumptions, we choose to include the keyword `sorry` when we state assumptions. Since our representation of proofs and assumptions are similar (because both are lemmas), it is important to understand this distinction.

```
lemma assumption:
  "[|A; B|] ==> C"
  sorry
```

Listing 2.7: An example of an assumption. We have `A` and `B`, but we cannot derive `C`. We defer proving the lemma using the Isabelle keyword `sorry`, which tells Isabelle to just accept our lemma as true without proof.

2.4 Language

In their work, Syeda and Klein present the semantics of WHILE [10], which is a language with support for privileged statements providing operating system functionality. The authors define statements for flushing the TLB, setting the page table root, setting the current process (which is a hardware feature of ARM), and setting the operating mode. The authors define the semantics of each statement and thereby provide a foundation to build a compiler. In addition to giving the semantics, the authors also give definitions and lemmas for reasoning about features such as expression evaluation and memory access (including page table walks).

Listing 2.8 shows how Seyda and Klein define arithmetic expressions in WHILE. There are constants, unary expressions, binary expressions, and heap (memory) lookups. As shown by the listing, unary and binary expressions are abstract, meaning no concrete operations are defined.

```
datatype aexp = Const val
              | UnOp "(val => val)" aexp
              | BinOp "(val => val => val)" aexp aexp
              | HeapLookup aexp
```

Listing 2.8: The data type `aexp`, defined by Seyda and Klein [10] and representing arithmetic expressions. The data type can have values consisting of `Const`, `UpOp`, `BinOp`, or `HeapLookup`. The values `UpOp` and `BinOp` take a function as argument, making them abstract. All values except `Const` take some other arithmetic expression as argument, making the arithmetic expressions recursive.

Listing 2.9 shows boolean expressions. There are constants, comparisons, binary expressions, and logical not. As shown by the listing, binary expressions are abstract, meaning no concrete operations are defined.

```
datatype bexp = BConst bool
              | BComp "(val => val => bool)" aexp aexp
              | BBinOp "(bool => bool => bool)" bexp bexp
              | BNot bexp
```

Listing 2.9: The data type `bexp`, defined by Seyda and Klein [10] and representing boolean expressions. The data type can have values consisting of `Const`, `BComp`, `BBinOp`, or `BNot`. The values `BBinOp` and `BNot` take a function as argument, making them abstract. All values except `Const` and `BComp` take some other boolean expression as argument, making the boolean expressions recursive.

Listing 2.10 shows statements, also referred to as commands. In addition to those described above, the authors also define statements for assignment (writing to memory), sequences (enabling programs consisting of more than one statement), conditional execution, and loops.

```
datatype com =  SKIP
              |  Assign aexp aexp       ("_ ::= _" [100, 61] 61)
              |  Seq com com            ("_;;/ _"  [60, 61] 60)
              |  If bexp com com        ("(IF _/ THEN _/ ELSE _)"  [0,
              ↪  0, 61] 61)
              |  While bexp com         ("(WHILE _/ DO _)"  [0, 61]
              ↪  61)
              |  Flush flush_type
              |  UpdateTTBR0 aexp
              |  UpdateASID asid
              |  SetMode mode_t
```

Listing 2.10: The data type `com`, defined by Seyda and Klein [10] and representing statements. We show the possible values in the listing. The values `Seq`, `If`, and `While` take some other statement as argument, making them recursive. The notation within parentheses is operator declarations, but since we will not use this notation in this report, we refrain from describing it further.

2.5 Compilers

A compiler [14] is a computer program that translates a source language into a target language. The most common translation is from a high-level computer language into machine language, but other translations are possible as well.

When using a compiler to translate a high-level computer language into machine code (a process called compilation), some significant advantages include making programs more readable to humans and making them portable between computer architectures. There are disadvantages as well, such as making it harder or impossible to verify that code behaves according to specification.

3
Methods

This chapter describes the general methods we use when implementing the compiler and stating the correctness proof. It describes the techniques in a top-down approach since that is the method we use both when developing the compiler as when stating the correctness theorem. We begin by describing how we reason about modifications to the source language semantics. Then, we explain how we develop the compiler itself. Since the Cambridge ISA (described in section 2.1) provides no mnemonics, we continue with describing the assembler that we produce to enable the use of mnemonics in the compiler. Finally, we describe our reasoning about the instruction cycle, since it is necessary to reason about it when verifying the correctness of the mnemonics emitted by the compiler.

Because we build upon previous work by Syeda and Klein [7] [10] in this project, and since the authors do that work using Isabelle, we choose to use Isabelle as well. One could argue that choosing Isabelle is a comfort selection. Still, since we find it infeasible to port the previous work to some other theorem prover, we decide not to investigate that possibility.

To simplify both the implementation and the correctness theorem, we make heavy use of definitions, functions, and lemmas. At all times, we try to find patterns, enabling us to reuse code and proof patterns. To avoid unnecessary dependencies, we also observe which layers are present and try to separate them as strictly as possible.

By reusing code and observing layers, we isolate features to clearly defined functions, giving the compiler a directed tree-like structure, where the root is the main compiler function (described in section 4.5.3.9). The root has child nodes consisting of functions handling statements, and in turn, the statement functions have child nodes consisting of expression and instruction functions. The expression functions have child nodes consisting of instructions, and this pattern continues down to the level of the instruction cycle.

We have one proof or assumption per function, and therefore, the structure of the proof follows the same pattern. The root of the proof is the compiler correctness theorem, which depends on proofs for the statement functions, which depends on proofs for the expression functions. This pattern continues down to the instruction cycle proofs, giving the proof the same directed tree-like structure as the compiler implementation.

There exist two distinct types of states which we identify. The first type is the program state, which is defined by the source language. The second one is the machine state, established by the processor. These two state types are related while not identical, and one design goal is to define this relation while keeping them separated in accordance with our layered design.

When proving properties (i.e., writing lemmas), we always aim to formulate as powerful conclusions as possible. In practice, this means we in first hand try to show that some operation preserves the state except for specific properties, and in second hand, prove that the operation leads to a new state with particular properties. As an example, consider Listing 3.1 and 3.2:

```
lemma IncPC_correct:
  "[|machine_config s;
    IncPC () s = ((),t)|] ==>
      t = s(|REG := (REG s)(RName_PC := REG s RName_PC + 4)|) &
      flags_preserved s t &
      machine_config t &
      machine_state_preserved s t"
```

Listing 3.1: An example of a lemma, `IncPC_correct`, proving that `IncPC` increments `PC` by 4. The proof states that the new state `t` equals the old state `s`, except for the `REG` field. We explain the replacement operators `(|` and `|)` in subsection 2.3.1.

```
lemma IncPC_correct2:
  "[|machine_config s;
    IncPC () s = ((),t)|] ==>
      REG t RName_PC = REG s RName_PC + 4 &
      flags_preserved s t &
      machine_config t &
      machine_state_preserved s t"
```

Listing 3.2: An example of a lemma, `IncPC_correct2`, proving that `IncPC` increments `PC` by 4. The proof states the value of `PC` and some other properties of state `t`, but the rest of the properties of state `s` are lost.

3.1 Language

Before implementing the compiler and stating the correctness proof, we reason about which existing language features are hard to prove, and what we need to add to the source language. One feature that would require extensive proving is binary expressions. Because of the recursive nature of expressions, there is a risk of running out of registers to store intermediate results. Usually, the compiler handles this case using

the stack. However, since using the stack implies accessing memory (complicating the verification significantly), we choose to redefine the language into only allowing constants as subexpressions.

Syeda and Klein define expressions abstractly [10] in their language, meaning there are no concrete operations such as addition or subtraction defined. To compile the expressions shown in subsection 2.4 into mnemonics, we need to specify at least one concrete operation per expression type. From a verification perspective, it is of less importance which operations we define, so we choose to add:

1. Negation, for unary arithmetic expressions
2. Addition and subtraction, for binary arithmetic expressions
3. And and Or, for binary boolean expressions

To add concrete operations, we change the source language (described in section 2.4) from specifying abstract operations using function types to specifying concrete operations using the data types in listing 3.3 to 3.6.

```
datatype aunop = Neg
```

Listing 3.3: The `aunop` definition. We specify the unary arithmetic expressions using this type.

```
datatype abinop = Plus | Minus
```

Listing 3.4: The `abinop` definition. We specify the binary arithmetic expressions using this type.

```
datatype bbinop = And | Or
```

Listing 3.5: The `bbinop` definition. We specify the binary boolean expressions using this type.

```
datatype bcomp = Less
```

Listing 3.6: The `bcomp` definition. We specify the comparison expressions using this type.

In addition to specifying the concrete operations, we also change the boolean negation operation specified in the source language to a unary boolean operation with a concrete negation operation, shown in listing 3.7. Our reason for this is to keep the expressions uniform and enable adding new unary boolean operations.

```
datatype bunop = Not
```

Listing 3.7: The `bunop` definition. We specify the unary boolean expressions using this type.

A final change we make to the source language before implementing the compiler is to only allow single addresses instead of ranges when using the flush statements defined by the source language. Our reasoning behind this is that the ARM architecture only allows for single addresses when using the required instructions. Therefore, allowing ranges would force the compiler to emit hidden loops.

3.2 Compiler

To facilitate the correctness theorem, we reason about which properties the compiler will have before implementing it. Our goal with this is to promote reusability and simplicity, since having features such as optimization will complicate the correctness theorem. Also, because of the changes to the source language described in section 3.1, we decide to implement all computation using registers only.

Patterson and Ditzel argue the case for a RISC computer [15]. In their work, the authors argue why using a RISC instruction set reduces the burden on the compiler developer. When we apply this reasoning to our project, we conclude that it is reasonable to reduce the number of instructions used to a minimum. Our goal is to produce a verified and not an optimal compiler, and hence, we design the compiler using as few instructions as possible.

When reasoning about the language described by Syeda and Klein [10], it is apparent that many of the statements require the processor executing in a privileged mode. Therefore, another decision we make when developing and verifying the compiler is to assume that the processor is running in kernel mode at the start of execution. There exist more properties belonging to the processor configuration, and we describe them further in subsection 4.1.1.

3.3 Assembler

An assembler is a computer program that translates a source language consisting of mnemonics and macros into machine code. A mnemonic is a textual representation of a machine code, which is more readable to humans. When we use the word mnemonic throughout this report, we always adhere to this definition.

There exist two principal reasons for us to implement an assembler in addition to our compiler. The first reason is that we need to reason about the instruction cycle, which expects to fetch machine words from memory. The second reason is conditional instructions, which are required to implement features such as if statements and loops, are not possible to express using the instruction type provided by

the Cambridge ISA.

When implementing our assembler, we use the ARM manual as ground truth, and then implement the assembler with support for each instruction emitted by the compiler.

The ARM architecture [9] categorizes the instructions into classes (such as Data-processing, Coprocessor) and then into subclasses (such as Data-processing (register), Data-processing (immediate)). The Cambridge ISA follows the same pattern, and when developing the assembler, the method we use is to take a given instruction and then look it up the class and subclass in the ARM ISA. Then, we define each mnemonic using the class and subclasses provided by the Cambridge ISA.

In addition to defining each mnemonic according to the ARM manual, the assembler also provides aliases for some instructions to improve readability.

3.4 Cambridge ISA

Even though most instructions used by the compiler are present in the Cambridge ISA [6], one particular instruction dealing with the coprocessor (described in subsection 4.4.7) is not. Concretely, this means that we are not able to implement support for operation requiring interfacing a coprocessor, such as flush, without modifying the Cambridge ISA.

The method we choose to address this is to carefully study the architecture of the Cambridge ISA [6], the ARM manual [9], and finally, the work on TLB consistency done by Syeda and Klein [7] [10]. Then, we add the required logic to the Cambridge ISA in accordance with our findings and conclusions from these previous works.

4

Results

This chapter describes the result of our implementation of the compiler and correctness theorem. Contrary to the previous chapter, this chapter presents the results in a bottom-up approach, beginning with the basic definitions describing the assumed machine configuration and the state relation.

After describing the basic definitions, we continue with a description of the instruction cycle. We continue with our description of the resulting assembler, and we give definitions and functions used by the compiler. We pair each described item with its corresponding proof, or, in case there is no such proof, clearly stated assumptions.

We work our way up the implementation until we reach the main compiler function, whose proof constitutes the compiler correctness theorem.

4.1 Basic definitions

This section describes the basic definitions that occur regularly throughout this chapter. We introduce each definition by name and describe its usage, together with derived lemmas, if applicable.

4.1.1 Machine configuration

Using the definition shown in Listing 4.1, we express that the processor has the correct configuration (as expected by the compiler) in a given state `s`. This definition introduces `Architecture`, `Encoding`, and `Extensions`, `Aligned1` which are definitions defined in the Cambridge ISA [6]. These definitions describe how the processor operates, and we only use them in this definition.

In summary, the definition states that the processor has the expected architecture, runs normal ARM instructions, that `PC` is aligned to 4 (i.e., the address is divisible by 4), runs in user or kernel mode, and that there is no exception raised. It is possible to prove that a given operation preserves the processor configuration by having this definition as an assumption, and then, as part of the conclusion.

```
definition
  machine_config :: "'a set_tlb_state_scheme => bool"
where
  "machine_config s = (
    Architecture s = ARMv7_A &
    Encoding s = Encoding_ARM &
    Extensions s = {} &
    Aligned1 (Addr (REG s RName_PC), 4) &
    ~J (CPSR s) & ~T (CPSR s) & ~E (CPSR s) &
    (PSR.M (CPSR s) = 0x10 | PSR.M (CPSR s) = 0x13) &
    exception s = NoException
  )"
```

Listing 4.1: The `machine_config` definition. The definition takes a machine state and returns a value of type `bool`. It is true in case all properties listed have the expected values. When we assume this definition, it means that we assume these properties.

4.1.2 Machine state preservation

We use the definition shown in Listing 4.2 to express that some operation preserves the subset of the machine state relevant to the source language. The definition is not usable when a statement intentionally changes some property of this subset. In those cases, we instead rely upon proving a subset of this definition.

This definition introduces `iset`, `global_set`, and `snapshot`, which are definitions defined by Syeda and Klein [7]. Their purpose is to describe the state of the TLB, but it suffices to compare them when used in this definition.

```
definition
  machine_state_preserved :: "'a set_tlb_state_scheme => 'a
  ↪  set_tlb_state_scheme => bool"
where
  "machine_state_preserved s t ==
    ASID s = ASID t &
    TTBR0 s = TTBR0 t &
    iset (set_tlb s) = iset (set_tlb t) &
    global_set (set_tlb s) = global_set (set_tlb t) &
    snapshot (set_tlb s) = snapshot (set_tlb t) &
    PSR.M (CPSR s) = PSR.M (CPSR t) &
    MEM s = MEM t"
```

Listing 4.2: The `machine_state_preserved` definition, taking two machine states `s` and `t`. The definition is true if all compared properties have the same values.

4.1.3 Code execution

We use the `steps` function whenever we want to express that the processor in a given state executes some instructions. Here, we define a step as a complete instruction cycle, i.e., we mean that the processor fetches, decodes, and runs some instructions. For example, we express the resulting state when from state `t` executing `k` steps as `steps t k = t'`, where `t'` is the resulting state.

```
fun
  steps :: "'a set_tlb_state_scheme => nat => 'a
  ↪  set_tlb_state_scheme"
where
  "steps s 0 = s" |
  "steps s (Suc i) = steps (snd (Next s)) i"
```

Listing 4.3: The `steps` function, taking a state `s` representing the current state and a natural number `i` representing the number of steps to take. There are two cases; if `i` is zero, then the function returns the current state. Otherwise, the function calls `Next` (described in section 4.2) and then executes a recursive call using the state returned from `Next`, and `i` decremented as arguments.

4.1.4 Reasoning about code installed in memory

We use the `code_installed` definition when we want to express that there are some instructions installed at the address pointed to by the program counter (PC) of a given state. With this, we mean that doing Fetch (and not the other steps of the instruction cycle) and then incrementing the PC for `k` times in state `t` would yield the first `k` instructions.

Listing 4.4 shows how we express this. When we say that some code is installed, then we mean that in some state `t` there are some instructions `mc` such that if `mc` fits in memory, and for all `i` steps where the PC is within `mc`, then Fetch yields the instruction pointed to by the PC after taking `i` steps.

```
definition
  code_installed :: "'a set_tlb_state_scheme => MachineCode list =>
  ↪  bool"
where
  "code_installed t mc == unat (REG t RName_PC) + 4 * length mc <
  ↪  2^32 &
                          (ALL i. let pci = REG t RName_PC;
                            pcf = REG (steps t i) RName_PC in pci =<
↪  pcf &
                            unat (pcf - pci) div 4 =< length mc -->
↪  (let (m, ft) = Fetch (steps t i) in exception ft = NoException &
                            m =  mc ! (unat (pcf - pci) div 4)))"
```

Listing 4.4: The `code_installed` definition, taking a machine state `t` and a list if machine codes `mc`. If this definition is true, then doing Fetch for some `k` times produce the first `k` machine codes in `mc`.

From this definition, we derive three important lemmas, shown in Listing 4.5, 4.6, and 4.7.

The lemma in Listing 4.5 states that if there is some state `t` with some instructions `a @ b` (i.e., some list `a` prepended to some list `b`, explained in subsection 2.3.1) installed, then `a` is installed in state `t`. This lemma enables us to reason about the instruction the processor will execute next.

```
lemma code_installed_append:
  "code_installed t (a @ b) ==> code_installed t a"
```

Listing 4.5: The `code_installed_append` proof. If `code_installed t (a @ b)` is true, then `code_installed t a` is also true.

Listing 4.6 shows a lemma which states that if there is some state `t` with some instructions `ca @ cb` installed and taking `k` steps results in PC pointing to `cb`, then `cb` is installed in state `steps t k`. This lemma enables us to reason about the instruction the processor will execute after `k` steps.

```
lemma code_installed_prepend:
  "[|code_installed t (ca @ cb);
    REG (steps t k) RName_PC = REG t RName_PC + 4 * (word_of_int
↪   (int (length ca)))|] ==>
    code_installed (steps t k) cb"
```

Listing 4.6: The `code_installed_prepend` proof. If `code_installed t (ca @ cb)` is true, and if `PC` points to `cb` after some `k` steps, then `code_installed (steps t k) cb` is true.

The lemma in Listing 4.7 states that if there is some state `t` with some instructions `x # xs` (i.e., some element `x` prepended to some list `xs`, explained in subsection 2.3.1) installed, then `Fetch t` will return `x` and some other state `snd (Fetch t)`.

```
lemma code_installed_Fetch:
  "code_installed t (x#xs) ==> Fetch t = (x, snd (Fetch t))"
```

Listing 4.7: The `code_installed_Fetch` proof. If `code_installed t (x#xs)` is true, then `Fetch x` will return `x` and some other state.

While these lemmas are essential for the compiler correctness, there are still lemmas missing. In particular, there is a need to formulate lemmas for reasoning about installed code treated as values. Without such lemmas, it is hard or impossible to prove instructions which loads values relative to the PC.

4.1.5 Reasoning about state relationship

We use the `state_rel` definition, shown in listing 4.8, to express that some program state relates to some machine state, bridging these state types. We use it in the compiler correctness proof, in which we assume that the state relationship holds before and conclude that it holds after executing each statement.

```
definition
  state_rel :: "p_state => 'a set_tlb_state_scheme => bool"
where
  "state_rel s t ==
     (asid s = ASID t) &
     (root s = TTBR0 t) &
     aligned (root s) &
     (incon_set s = iset (set_tlb t)) &
     (p_state.global_set s = global_set (set_tlb t)) &
     (ptable_snapshot s = snapshot (set_tlb t)) &
     mode_rel (mode s) (PSR.M (CPSR t)) &
     heap_rel s t"
```

Listing 4.8: The `state_rel` definition, taking one machine state `s` and one program state `t`. The definition is true if all compared properties have the same values.

At this point, we want to draw attention to the similarity between `state_rel` and `machine_state_preserved` (described in subsection 4.1.2). The reason for this similarity is that both these definitions describe the state relationship but in different contexts. We have this distinction because it does not make sense to reason about the program (i.e., source language) state at the assembler level.

One could argue that our state relationship is unexpectedly simple. The reason for this, as we interpret it, is that Syeda and Klein construct an ARM-style MMU model [7], and then define the source language [10] accordingly. The result is a state relationship with a one-to-one correspondence between machine and source language.

4.1.6 Converting binary values to register names

A final definition that occurs regularly through this chapter is the `bin_to_reg` definition, which we show in listing 4.9. The reason for having this definition is that the Cambridge ISA uses a mix of binary values and symbols when referring to registers. It is therefore sometimes necessary to convert between these data types.

```
definition
  bin_to_reg :: "4 word => RName"
where
  "bin_to_reg r = (
    if r = 0 then RName_0usr
    else if r = 1 then RName_1usr
    else if r = 2 then RName_2usr
    else if r = 3 then RName_3usr
    else if r = 4 then RName_4usr
    else if r = 5 then RName_5usr
    else if r = 6 then RName_6usr
    else if r = 7 then RName_7usr
    else if r = 8 then RName_8usr
    else if r = 9 then RName_9usr
    else if r = 10 then RName_10usr
    else if r = 11 then RName_11usr
    else if r = 12 then RName_12usr
    else if r = 13 then RName_SPusr
    else if r = 14 then RName_LRusr
    else RName_PC
  )"
```

Listing 4.9: The `bin_to_reg` definition, taking a four-bit word and returning a register name. There are sixteen possible values, which are all covered by this definition.

4.2 Instruction cycle

In this section, we describe our findings about the instruction cycle in more detail. Even though the compiler proofs only use the function `steps` directly, `steps` depends on the Cambridge ISA function `Next`, which in turn depends on functions representing each stage of the instruction cycle. In addition, the assembler proofs deal with the Fetch stage directly.

4.2.1 Fetch

In the Fetch stage, the processor reads the next instruction (in the form of machine code) from a location in memory pointed to by the PC. If the processor is unable to read from memory (for instance, if there is no mapping for the address in the PC), this stage might fail, causing an exception.

Even though we did not have to make any modifications to the Fetch stage in the Cambridge ISA, we still had to prove that it does preserve the machine config, the machine state, and the registers. We show our correctness proof for this in Listing 4.10.

```
lemma Fetch_correct:
  "[|Fetch s = (mc, t);
    machine_config s|] ==>
      flags_preserved s t &
      machine_config t &
      machine_state_preserved s t &
      REG s = REG t"
```

Listing 4.10: The `Fetch_correct` proof. We show that a successful Fetch preserves the flags, the machine configuration, the machine state, and the registers.

4.2.2 Decode

In the Decode stage, the processor interprets the machine code read in the previous step as an instruction. This stage also handles conditional execution by returning different instructions depending on the condition code field (the first four bits of each machine word) and the state of the processor flags. In case the flags match the condition, this stage decodes and returns some instruction determined by the rest of the machine word. Otherwise, it returns a no-operation (NOP) instruction, which only increments the PC by 4.

Listing 4.11 shows an example of how we define one instance of the condition code field. There are a total of 15 conditions defined in the ARM architecture [9], and except for this example, we explain conditions when they occur in this report. For our example, we choose the `always` condition since it is the most common by far. The definition takes 28 bits as input and returns a machine code, where the 28 bits are the instruction part itself.

```
definition
  always :: "28 word => MachineCode"
where
  "always m = ARM (word_cat (0xe::4 word) m)"
```

Listing 4.11: The `always` definition. The definition takes a 28-bit word consisting of the lower bits of the machine code and returns a machine code. It creates the machine code by concatenating the 28-bit word to the 4-bit word that represents the conditional code always.

We investigated the possibility of proving the Decode stage for each mnemonic defined by the assembler, but that turned out to be infeasible. The reason for this is that the Cambridge ISA uses a switch statement matching binary patterns when decoding machine words, which is computationally expensive to prove. Therefore, we decided to rely upon close inspection of the Cambridge ISA's source code in combination with traditional testing instead of proofs for our mnemonics.

4.2.3 Run

In the Run stage, the processor executes the instruction returned from the previous step. Since the behavior is highly dependent upon the instruction in question, we do not reason about the Run stage in general. Instead, we reason about each instruction individually in section 4.4.

4.3 Cambridge ISA

Our modifications to the Cambridge ISA consist of adding the definition `dfn'MoveToCoprocessorFromRegister` (shown in listing 4.12), and modifying the existing logic to use our definition when receiving an `MCR` instruction (described in subsection 4.4.7).

The definition `dfn'MoveToCoprocessorFromRegister` is large compared to other definitions, but at the same time, it is not complex. It takes a tuple containing the operands of `MCR`, a machine state, and returns a new machine state. Interpreting the operands in accordance with the specification of the ARM ISA [9], it decodes the requested operation and passes the relevant data to the correct function from previous work by Syeda and Klein [7]. Therefore, our modifications to the Cambridge ISA act as an adapter, bridging our assembler layer (and indirectly, the compiler layer) with the previous work.

Having a definition that works as an adapter, assuming that our decoding of the operands is correct, enables us to conclude that our `MCR` mnemonic has the same semantics as the functions provided by the previous work. However, one should note that our adapter layer at the time of writing only supports the flush operations provided by Syeda and Klein.

```
definition dfn'MoveToCoprocessorFromRegister ::
  "3 word × 4 word × 4 word × 4 word × 3 word × 4 word => 'b::mmu
  ↪  state_scheme => unit × 'b::mmu state_scheme"
where
  "dfn'MoveToCoprocessorFromRegister == %(opc1, crn, rt, coproc,
  ↪  opc2, crm). do {
     v <- CurrentModeIsNotUser ();
    if v then (
      if opc1 = 0 then flush FlushTLB
      else if opc1 = 1 then do {
        reg <- read_state REG;
        value <- return(reg RName_0usr);
        asid <- return(word_extract 31 24 value);
        addr <- return(word_extract 23 0 value);
        flush (FlushASIDvarange asid {(Addr addr)::vaddr})
      }
      else if opc1 = 2 then do {
        reg <- read_state REG;
        asid32 <- return(reg RName_0usr);
        asid <- return(ucast asid32::8 word);
        flush (FlushASID asid)
      }
      else if opc1 = 3 then do {
        reg <- read_state REG;
        addr <- return(reg RName_0usr);
        flush (Flushvarange {(Addr addr)::vaddr})
      }
      else raise'exception (IMPLEMENTATION_DEFINED HOL.undefined)
    )
    else raise'exception (IMPLEMENTATION_DEFINED HOL.undefined)
  }"
```

Listing 4.12: The `dfn'MoveToCoprocessorFromRegister` definition, taking a tuple consisting of six operands and machine state. The definition interprets the operands and passes control to the `flush` function defined by Syeda and Klein, and thereby returns the expected state.

4.4 Assembler

Our resulting assembler supports the mnemonics described in this section. We defined each mnemonic to reassemble the instructions described in the ARM manual [9] as closely as possible.

Since we were developing a non-optimizing compiler, there was no general need for mnemonics to modify condition flags, which we explain in subsection 2.2.3. As a

consequence, we will only mention the flags when describing instruction changing or relying upon them. Similar reasoning applies to PC. All instructions defined by us, except for the branch instruction, increment PC by 4. Hence, we will not generally describe PC when describing an instruction and its proof.

In this subsection, many definitions use helper definitions. These are definitions having the same name as as the definition using them, except having no conditional naming and ending with 1. For example, add_reg1 is a helper function of add_reg, and mov_imm1 is a helper function of mov_imm and moveq_imm. The only purpose of helper functions is to concatenate bit strings, providing the 28 bits encoding the instruction specific part of machine codes, enabling reuse of this part when defining conditional instructions. We refrain from showing helper definitions, and instead refer the ARM ISA Reference Manual [9] for information about the encoding of instructions.

4.4.1 ADD

The ADD instruction takes three register operands. When executing this instruction, the processor reads the terms from the registers rn and rm, adds them, and stores the resulting sum in rd. We show how we define the mnemonic in listing 4.13.

```
definition
  add_reg :: "4 word => 4 word => 4 word => MachineCode"
where
  "add_reg rd rn rm = always (add_reg1 rd rn rm)"
```

Listing 4.13: The add_reg definition. The definition takes a 4-bit word rd, a 4-bit word rn, a 4-bit word rm and returns the machine code corresponding to an unconditional ADD instruction.

We proved the correctness of the ADD mnemonic, and we show the resulting lemma in Listing 4.14. We assume that all registers used are general-purpose (explained in subsection 2.2.3), and we conclude that after executing this instruction, rd contains the sum of the values contained in rn and rm before executing it.

```
lemma add_reg_correct:
  "[|Fetch t = (add_reg rd rn rm, ft);
    general_purpose_reg rd;
    general_purpose_reg rn;
    general_purpose_reg rm;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rd := REG t (bin_to_reg rn) +
↪  REG t (bin_to_reg rm),
                         RName_PC := REG t RName_PC + 4)"
```

Listing 4.14: The `add_reg_correct` proof. We show that executing an `ADD` instruction with expected operands produces the expected value in `rd`, and preserves the machine state.

4.4.2 AND

The `AND` instruction takes three register operands. When executing this instruction, the processor reads the values from the registers `rn` and `rm`, performs a bitwise and on them, and stores the result in `rd`. We show how we define the mnemonic in listing 4.15.

```
definition
  and_reg :: "4 word => 4 word => 4 word => MachineCode"
where
  "and_reg rd rn rm = always (and_reg1 rd rn rm)"
```

Listing 4.15: The `and_reg` definition. The definition takes a 4-bit word `rd`, a 4-bit word `rn`, a 4-bit word `rm` and returns the machine code corresponding to an unconditional `AND` instruction.

We proved the correctness of the `AND` instruction, and we show the resulting lemma in Listing 4.16. We assume that all registers used are general purpose, and we conclude that after executing this instruction, `rd` contains the bitwise and of `rn` and `rm` before executing it.

```
lemma and_reg_correct:
  "[|Fetch t = (and_reg rd rn rm, ft);
    general_purpose_reg rd;
    general_purpose_reg rn;
    general_purpose_reg rm;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rd := REG t (bin_to_reg rn) &&
↪  REG t (bin_to_reg rm),
                          RName_PC := REG t RName_PC + 4)"
```

Listing 4.16: The `and_reg_correct` proof. We show that executing an `AND` instruction with expected operands produces the expected value in `rd`, and preserves the machine state.

4.4.3 B

The `B` instruction takes an immediate value `imm12` as an operand. When executing this instruction, the processor adds `imm12` to `PC`, meaning that the processor performs a branch to some location in memory. We show how we define the mnemonic in listing 4.17.

```
definition
  b_imm :: "24 word => MachineCode"
where
  "b_imm imm24 = always (b_imm1 imm24)"
```

Listing 4.17: The `b_imm` definition. The definition takes a 24-bit word `imm24`, and returns the machine code corresponding to an unconditional `B` instruction.

Listing 4.18 shows the proof of the `B` instruction. We assume that `offset` is aligned with 4 (since the two least significant bits are zero), and conclude that the instruction adds `offset` to `PC`.

```
lemma b_imm_correct:
  "[|Fetch t = (b_imm offset, ft);
    machine_config t;
    word_extract 1 0 offset = (0::2 word)|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(RName_PC := REG t RName_PC + (ucast offset)
↪   + 8)"
```

Listing 4.18: The `b_imm_correct` proof. We show that executing a `B` instruction with `offset` aligned with 4 preserves the machine configuration, the machine state, and adds the offset to the `PC`.

We also define a conditional variant of `B` executing in case the `Z` flag (described in subsection 2.2.3) being set. We show the definition in listing 4.19

```
definition
  beq_imm :: "24 word => MachineCode"
where
  "beq_imm imm24 = eq (b_imm1 imm24)"
```

Listing 4.19: The `beq_imm` definition. The definition takes a 24-bit word `imm24`, and returns the machine code corresponding to a conditional `B` instruction, executing if the `Z` flag is set.

The proof of the conditional variant is analogous to the unconditional. The proof of both variants relies upon the assumptions in listing 4.20 and 4.21, but since is obvious that these assumptions are true, we still consider the branch instruction proved.

```
lemma Aligned1_assumption:
  "word_extract 1 0 offset = (0::2 word) ==>
     Aligned1 (Addr (word_cat (word_extract 31 2 (REG s RName_PC + 8
↪   + UCAST(24 -> 32) offset)) 0), 4)"
  sorry
```

Listing 4.20: The `Aligned1_assumption` lemma. We assume that if the last two bits of `offset` is zero (i.e., `offset` is divisible by 4), then `PC + offset` is aligned. The reason for this is that `PC` is divisible by 4 (see subsection 4.1.1), and that if an address is divisible by 4, then it is aligned.

```
lemma word_cat_assumption:
  "word_extract 1 0 offset = (0::2 word) ==>
     word_cat (word_extract 31 2 (REG s RName_PC + 8 + UCAST(24 ->
↪  32) offset)) 0 =
     REG s RName_PC + UCAST(24 -> 32) offset + 8"
  sorry
```

Listing 4.21: The `word_cat_assumption` lemma. We use the same reasoning as for listing 4.20 when we assume the correctness of this lemma.

4.4.4 CMP

The `CMP` instruction takes two operands, one register `rn`, and one immediate value `imm12`. When executing this instruction, the processor subtracts `imm12` from `rn`, sets the flags, and discards the result. We show how we define the mnemonic in listing 4.22.

```
definition
  cmp_imm :: "4 word => 12 word => MachineCode"
where
  "cmp_imm rn imm12 = always (data_arith_logic_imm 0xa 1 rn 0
  ↪  imm12)"
```

Listing 4.22: The `cmp_imm` definition. The definition takes a 4-bit word `rn`, a 12-bit word `imm12`, and returns the machine code corresponding to an unconditional `CMP` instruction. The hard-coded values passed to the helper definition `data_arith_logic_imm` are values that encodes a `CMP` instruction.

We proved the correctness of the `CMP` instruction, and we show the resulting lemma in Listing 4.23. Since we only use the `CMP` instruction when implementing boolean expressions (described in subsection 4.5.2), we assume that `reg` has the value zero if `val` is false, and one if `val` is true.

```
lemma cmp_imm_correct:
  "[|Fetch t = (cmp_imm reg 0, ft);
    REG t (bin_to_reg reg) = (if val then 1 else 0);
    general_purpose_reg reg;
     machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        PSR.Z (CPSR t') = (~val) &
        REG t' = (REG t)(RName_PC := REG t RName_PC + 4)"
```

Listing 4.23: The `cmp_imm_correct` proof. We show that executing a `CMP` instruction with expected operands preserves the machine configuration, and the machine state. If `val` is false, then the instruction sets the `Z` flag. Otherwise, the instruction clears the `Z` flag.

4.4.5 LDR (immediate)

The `LDR (immediate)` instruction takes two register operands and one immediate value. When executing this instruction, the processor computes an address by adding the value in `rn` with `imm12`, reads a 32-bit word from the computed address, and then stores the word read in `rd`. We show how we define the mnemonic in listing 4.24.

```
definition
  ldr_imm :: "4 word => 4 word => 12 word => MachineCode"
where
  "ldr_imm rt rn imm12 = always (ldr_imm1 0 0 0 rt rn imm12)"
```

Listing 4.24: The `ldr_imm` definition. The definition takes a 4-bit word `rt`, 4-bit word `rn`, a 12-bit word `imm12`, and returns the machine code corresponding to an unconditional `LDR` instruction. The hard-coded values passed to the helper definition are values that encodes an `LDR` instruction that does not modify values in the register operands.

While we do not prove the correctness of the `LDR` instruction (because of an issue related to alignment), we show our assumption in Listing 4.25. We assume that if `rn` and `rt` are general-purpose registers, and reading from the address in `rn` yields some value `val`, then `rt` will contain `val` after executing the instruction. To express that reading `rn` in state `t` yields `val`, we use the function `mmu_read_size`, defined by Syeda and Klein [7]. If this function returns `val` and some state `mt` with no exception, then `val` is readable from `rn` in state `t`.

```
lemma ldr_imm_correct:
  "[|Fetch t = (ldr_imm rt rn 0, ft);
    general_purpose_reg rn;
    general_purpose_reg rt;
    machine_config t;
    mmu_read_size (Addr (REG t (bin_to_reg rn)), 4) t = (to_bl val,
↪   mt);
    exception mt = NoException|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rt := val,
                         RName_PC := REG t RName_PC + 4)"
  sorry
```

Listing 4.25: The `ldr_imm_correct` assumption. Assuming that reading the address in `rn` yields some value `val`, we show that executing a `LDR` instruction with expected operands preserves the machine configuration, the machine state, and stores `val` in `rt`.

4.4.6 LDR (literal)

The `LDR (literal)` instruction takes one immediate bit `u`, one register operand `rt`, and one immediate value `imm12`. When executing this instruction, the processor computes an address by adding the value in `rt` with `imm12` if `u` is zero, and subtracts the value in `rt` with `imm12` if `r` is one. Then, the processor reads a 32-bit word from the computed address and then stores the word read in `rd`. Having this instruction enables us to read values from addresses relative to the `PC`, i.e., loading values that are part of the program. We show how we define the mnemonic in listing 4.26.

```
definition
  ldr_lit :: "1 word => 4 word => 12 word => MachineCode"
where
  "ldr_lit u rt imm12 = always (ldr_lit1 u rt imm12)"
```

Listing 4.26: The `ldr_lit` definition. The definition takes a bit `u`, a 4-bit word `rt`, a 12-bit word `imm12`, and returns the machine code corresponding to an unconditional `LDR` instruction.

Because we lack the lemmas necessary to reason about values that are part of the code installed in memory, we do not provide a correctness lemma for this instruction. Instead, when using this instruction (described in subsection 4.5.1), we provide our correctness assumption in the compiler layer.

4.4.7 MCR

The MCR instruction provides functionality to move data from the ARM core registers to a coprocessor (described in subsection 2.2.2). We show the definition in listing 4.40, and it takes a 3-bit word opc1, a 4-bit word crn, a register rt, a 4-bit word coproc, a 3-bit word opc2, and a 4-bit word crm. Except for rt, it is best to think of these operands as a way to point out a function in a specific coprocessor determined by coproc. It is best to think about the value contained in rt as an argument to the function.

```
definition
  mcr_reg :: "3 word => 4 word => 4 word => 4 word => 3 word => 4
  ↪  word => MachineCode"
where
  "mcr_reg opc1 crn rt coproc opc2 crm = always (mcr_reg1 opc1 crn
  ↪  rt coproc opc2 crm)"
```

Listing 4.27: The mcr_reg definition. The definition takes the described operands, and returns the machine code corresponding to an unconditional MCR instruction.

While we provide the MCR instruction as a mnemonic, we only use a small number of coprocessor functions in this project. Therefore, to improve readability, we do not use the MCR instruction directly in our compiler. Instead, we rely upon aliases, which we describe in this subsection. For a complete description of the MCR instruction and the coprocessor concept, we refer to the ARM ISA Reference Manual [9].

4.4.7.1 SETASID

Listing 4.28 shows the SETASID mnemonic, an alias for MCR with operands set to modify the current process (described in subsection 2.2.3) of the processor.

```
definition
  setasid :: "4 word => MachineCode"
where
  "setasid rt = mcr_reg 0 13 rt 15 0 0"
```

Listing 4.28: The setasid definition. The definition takes a register rt, and returns the machine code corresponding to an unconditional MCR instruction with the operands hard-coded to change the current process using the value in rt as argument.

Because the underlying implementation of MCR lacks support for updating the process id (described in section 4.3), we refrain from giving our proof assumption in the assembler layer. Instead, we give our proof assumption for updating the process id in the compiler layer (described in section 4.5).

4.4.7.2 SETTTBR0

Listing 4.29 shows the `SETTTBR0` mnemonic, an alias for `MCR` with operands set to modify the page table root (described in subsection 2.2.3) of the processor.

```
definition
  setttbr0 :: "4 word => MachineCode"
where
  "setttbr0 rt = mcr_reg 0 2 rt 15 0 0"
```

Listing 4.29: The `setttbr0` definition. The definition takes a register `rt`, and returns the machine code corresponding to an unconditional `MCR` instruction with the operands hard-coded to change the page table root using the value in `rt` as argument.

Because the underlying implementation of `MCR` lacks support for updating the page table root, we refrain from giving our proof assumption in the assembler layer. Instead, we give our proof assumption for updating the process id in the compiler layer.

4.4.7.3 TLBIALL

Listing 4.30 shows the `TLBIALL` mnemonic, an alias for `MCR` with operands set to flush all entries in the TLB.

```
definition
  tlbiall :: "MachineCode"
where
  "tlbiall = mcr_reg 0 8 0 15 0 7"
```

Listing 4.30: The `tlbiall` definition. The definition returns the machine code corresponding to an unconditional `MCR` instruction with the operands hard-coded to flush all entries in the TLB.

While we do not prove the correctness of the `TLBIALL` mnemonic, we show our assumption in listing 4.31. We assume that the processor runs in kernel mode (described in subsection 2.2.3), and since `MCR` only passes operands to the functions defined by Syeda and Klein [7] (described in section 4.3), we conclude that the instruction flushes the TLB in accordance with the semantics of their `flush` function.

```
lemma tlbiall_correct:
  "[|Fetch t = (tlbiall, ft);
    PSR.M (CPSR t) = 0x13;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        ASID t = ASID t' &
        TTBR0 t = TTBR0 t' &
        PSR.M (CPSR t) = PSR.M (CPSR t') &
        MEM t = MEM t' &
        global_set (set_tlb t') = Un (entry_op_class.range_of `
↪  TLB_ASIDs.global_entries (the ` {e E range (TLB.pt_walk (ASID t)
↪  (MEM t) (TTBR0 t)). ¬ is_fault e})) &
        iset (set_tlb t') = {} &
        snapshot (set_tlb t') = (\%a. ({}, \%v. Fault))"
    sorry
```

Listing 4.31: The `tlbiall_correct` assumption. We show that executing an `MCR` instruction with expected operands flushes all entries in the TLB, and preserves the rest of the machine state.

4.4.7.4 TLBIASID

Listing 4.30 shows the `TLBIASID` mnemonic, an alias for `MCR` with operands set to flush all entries in the TLB belonging to a given process.

```
definition
  tlbiasid :: "4 word => MachineCode"
where
  "tlbiasid rt = mcr_reg 0 8 rt 15 2 7"
```

Listing 4.32: The `tlbiasid` definition. The definition returns the machine code corresponding to an unconditional `MCR` instruction with the operands hard-coded to flush all entries in the TLB belonging to a given process.

4.4.7.5 TLBIMVAA

Listing 4.30 shows the `TLBIMVAA` mnemonic, an alias for `MCR` with operands set to flush all entries in the TLB belonging to a given address.

```
definition
  tlbimvaa :: "4 word => MachineCode"
where
  "tlbimvaa rt = mcr_reg 0 8 rt 15 3 7"
```

Listing 4.33: The `tlbimvaa` definition. The definition returns the machine code corresponding to an unconditional `MCR` instruction with the operands hard-coded to flush all entries in the TLB belonging to a given address.

4.4.7.6 TLBIMVA

Listing 4.30 shows the `TLBIMVA` mnemonic, an alias for `MCR` with operands set to flush all entries in the TLB belonging to a given address and process.

```
definition
  tlbimva :: "4 word => MachineCode"
where
  "tlbimva rt = mcr_reg 0 8 rt 15 1 7"
```

Listing 4.34: The `tlbimva` definition. The definition returns the machine code corresponding to an unconditional `MCR` instruction with the operands hard-coded to flush all entries in the TLB belonging to a given address and process.

4.4.8 MOV (immediate)

The `MOV (immediate)` instruction takes two operands, one register `rd`, and one immediate value `imm12`, which is copied to `rd` when executing the instruction. We show how we define the mnemonic in listing 4.35.

```
definition
  mov_imm :: "4 word => 12 word => MachineCode"
where
  "mov_imm rd imm12 = always (mov_imm1 rd imm12)"
```

Listing 4.35: The `mov_imm` definition. The definition takes a 4-bit word `rd`, a 12-bit word `imm12`, and returns the machine code corresponding to an unconditional `MOV` instruction with the given operands.

We proved the correctness of the `MOV` instruction, and we show the resulting lemma in Listing 4.36. If `reg` is general-purpose and if `val` is less than 256 (because the most significant bits are zero), then we conclude that after executing this instruction, `reg` contains `val`.

```
lemma mov_imm_correct:
  "[|Fetch t = (mov_imm reg val, ft);
    general_purpose_reg reg;
    machine_config t;
    word_extract 11 8 val = (0::4 word)|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg reg := ucast val,
                         RName_PC := REG t RName_PC + 4)"
```

Listing 4.36: The `mov_imm_correct` proof. We show that executing a `MOV` instruction with expected operands copies `val` to `rd`, and preserves the machine state.

We also define two conditional variants of `MOV`. The first variant, named `MOVEQ`, writes the value to `reg` in case of `Z` flag (described in subsection 2.2.3) is set. The second variant, named `MOVNE`, writes to `reg` in case of the `Z` flag is clear. We proved both variants while only showing the first one in listing 4.37. The proof is very similar to Listing 4.36, with the exception that we preserve the flags, enabling a pattern of an instruction setting the flags followed by one or more conditional instructions.

```
lemma moveq_imm_correct:
  "[|Fetch t = (moveq_imm reg val, ft);
    general_purpose_reg reg;
    machine_config t;
    word_extract 11 8 val = (0::4 word)|] ==>
      EX t'. steps t 1 = t' &
        flags_preserved t t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg reg := (if PSR.Z (CPSR t) then
↪  ucast val else REG t (bin_to_reg reg)),
                         RName_PC := REG t RName_PC + 4)"
```

Listing 4.37: The `moveq_imm_correct` proof. We show that executing a `MOV` instruction with expected operands preserves the flags, the machine configuration, and the machine state. In case the `Z` flag is set, the instruction also copies `val` to `rd`.

4.4.9 MOV (register)

The `MOV (register)` instruction takes two register operands `rd` and `rm`. Executing the instruction copies the value in `rm` to `rd`. We show how we define the mnemonic in listing 4.38.

```
definition
  mov_reg :: "4 word => 4 word => MachineCode"
where
  "mov_reg rd rm = always (mov_reg1 rd rm)"
```

Listing 4.38: The `mov_reg` definition. The definition takes a 4-bit word `rd`, a 4-bit word `rm`, and returns the machine code corresponding to an unconditional `MOV` instruction with the given operands.

We proved the correctness of the `MOV` instruction, and we show the resulting lemma in Listing 4.39. If `rd` and `rm` are general-purpose, then we conclude that this instruction copies the value in `rm` to `rd`.

```
lemma mov_reg_correct:
  "[|Fetch t = (mov_reg rd rm, ft);
    general_purpose_reg rd;
    general_purpose_reg rm;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rd := REG t (bin_to_reg rm),
                         RName_PC := REG t RName_PC + 4)"
```

Listing 4.39: The `mov_reg_correct` proof. We show that executing an `MOV` instruction with expected operands copies the value in `rm` to `rd` while preserving the machine state.

4.4.10 MSR

The `MSR` instruction provides a way to move data from the ARM core registers to `CPSR` register, which we describe in subsection 2.2.3. We show the definition in listing 4.40, and it takes a bit `r`, a 4-bit word `m`, and a register `rn`.

When the processor executes this instruction, it will write the low byte in the register `rn` to a byte determined by `m` in some register determined by `r`. We only use this instruction for writing to the `M` field of the `CPSR` register in this project, and therefore, we always set the operands accordingly. For more information on the operands for this instruction, we refer to the ARM ISA Reference Manual [9].

```
definition
  msr_reg :: "1 word => 4 word => 4 word => MachineCode"
where
  "msr_reg r m rn = always (msr_reg1 r m rn)"
```

Listing 4.40: The `msr_reg` definition. The definition takes a bit `r`, a 4-bit word `m`, a 4-bit word `rn`, and returns the machine code corresponding to an unconditional `MSR` instruction with the given operands.

While providing the `MSR` instruction (shown in Listing 4.40) as a mnemonic, we only (for reasons explained in subsection 4.5.3.8, we only use it to change processor mode from kernel to user in this project. Therefore, our correctness assumption (shown in Listing 4.41) assumes that we use the mnemonic for this purpose only. Consequently, we assume hard-coded operands, that the processor runs in kernel mode (described in subsection 2.2.3), and that the value of `R0` equals 0x10.

```
lemma msr_reg_correct:
  "[|Fetch t = (msr_reg 0 0x1 0, ft);
    PSR.M (CPSR s) = 0x13;
    REG t RName_0usr = 0x10;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        ASID t = ASID t' &
        TTBR0 t = TTBR0 t' &
        iset (set_tlb t) = iset (set_tlb t') &
        global_set (set_tlb t) = global_set (set_tlb t') &
        snapshot (set_tlb t) = snapshot (set_tlb t') &
        PSR.M (CPSR t') = 0x10 &
        MEM t = MEM t' &
        REG t' = (REG t)(RName_PC := REG t RName_PC + 4)"
  sorry
```

Listing 4.41: The `msr_reg_correct` assumption. We show that executing an `MSR` instruction with expected operands changes to processor mode from kernel to user mode, and preserves the rest of the machine state.

4.4.11 OR

The `OR` instruction takes three register operands. When executing this instruction, the processor reads the values from the registers `rn` and `rm`, performs a bitwise or on them, and stores the result in `rd`. We show how we define the instruction in listing 4.42.

```
definition
  or_reg :: "4 word => 4 word => 4 word => MachineCode"
where
  "or_reg rd rn rm = always (or_reg1 rd rn rm)"
```

Listing 4.42: The `or_reg` definition. The definition takes a 4-bit word `rd`, a 4-bit word `rn`, a 4-bit word `rm` and returns the machine code corresponding to an unconditional `OR` instruction.

We proved the correctness of the `OR` instruction, and we show the resulting lemma in Listing 4.43. We assume that all registers used are general purpose, and we conclude that after executing this instruction, `rd` contains the bitwise or of `rn` and `rm` before executing it.

```
lemma or_reg_correct:
  "[|Fetch t = (or_reg rd rn rm, ft);
    general_purpose_reg rd;
    general_purpose_reg rn;
    general_purpose_reg rm;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rd := REG t (bin_to_reg rn) ||
↪  REG t (bin_to_reg rm),
                         RName_PC := REG t RName_PC + 4)"
```

Listing 4.43: The `or_reg_correct` proof. We show that executing an `OR` instruction with expected operands produces the expected value in `rd`, and preserves the machine state.

4.4.12 RSB

The `RSB` instruction takes three operands, two register operands, and one immediate value. When executed, the instruction subtracts `rn` from `imm12` and stores the result in `rd`.

```
definition
  rsb_imm :: "4 word => 4 word => 12 word => MachineCode"
where
  "rsb_imm rd rn imm12 = always (rsb_imm1 rn rd imm12)"
```

Listing 4.44: The `rsb_imm` definition. The definition takes a 4-bit word `rd`, a 4-bit word `rn`, a 12-bit word `imm12`, and returns the machine code corresponding to an unconditional `RSB` instruction with the given operands.

Since we use this instruction to implement negation, we define an alias `NEG` to improve readability. We show this alias in listing 4.45. When executing this instruction, the processor reads the term from the register `rm`, subtracts it from zero, and stores the resulting negation in `rd`.

```
definition
  neg :: "4 word => 4 word => MachineCode"
where
  "neg rd rm = rsb_imm rd rm 0"
```

Listing 4.45: The `neg` definition. The definition takes a 4-bit word `rd`, a 4-bit word `rn` and returns the machine code corresponding to an unconditional `RSB` instruction with the given operands, and with `imm12` hard-coded to zero.

We proved the correctness of the `NEG` mnemonic, except for one subgoal (asking us to prove two's complement arithmetic), which we have left as an assumption. We show the resulting lemma in Listing 4.46. We assume that all registers used are general-purpose, and we conclude that after executing this instruction, `rd` contains the negation of `rn` before executing the instruction.

```
lemma neg_correct:
  "[|Fetch t = (neg rd rm, ft);
    general_purpose_reg rd;
    general_purpose_reg rm;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rd := -(REG t (bin_to_reg rm)),
↪  RName_PC := REG t RName_PC + 4)"
```

Listing 4.46: The `neg_correct` proof. We show that executing an `NEG` instruction with expected operands produces the expected value in `rd`, and preserves the machine configuration and the machine state.

4.4.13 STR

The `STR` instruction takes two register operands and one immediate value. When executing this instruction, the processor computes an address by adding the value in `rn` with `imm12`, reads 32 bits from `rt`, and then stores the result to the computed address.

```
definition
  str_imm :: "4 word => 4 word => 12 word => MachineCode"
where
  "str_imm rt rn imm12 = always (str_imm1 0 0 0 rt rn imm12)"
```

Listing 4.47: The `str_imm` definition. The definition takes a 4-bit word `rt`, 4-bit word `rn`, a 12-bit word `imm12`, and returns the machine code corresponding to an unconditional `STR` instruction. The hard-coded values passed to the helper definition are values that encodes an `STR` instruction that does not modify values in the register operands.

While we do not prove the correctness of the `STR` instruction, we show our assumption in Listing 4.48. We assume that if `rn` and `rt` are general-purpose registers and the address stored in `rn` is accessible (i.e., accessing it does not result in an exception), then the memory address in `rn` will contain `val` after executing the instruction. To express that writing to the address in `rn` in state `t` does not cause an exception, we use the function `mmu_write_size`, defined by Syeda and Klein [7]. If this function returns some state `mt` with no exception, then the address in `rn` is writable in state `t`.

```
lemma str_imm_correct:
  "[|Fetch t = (str_imm rt rn 0, ft);
    general_purpose_reg rn;
    general_purpose_reg rt;
    machine_config t;
    mmu_write_size (to_bl (REG t (bin_to_reg rt)), Addr (REG t
↪   (bin_to_reg rn)), 4) t = (x, mt);
    exception mt = NoException|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        mmu_read_size (Addr (REG t (bin_to_reg rn)), 4) t' = (to_bl
↪   (REG t (bin_to_reg rt)), t'')"
  sorry
```

Listing 4.48: The `str_imm_correct` assumption. Assuming that writing `val` to the address in `rn` does not raise an exception, we show that executing a `STR` instruction with expected operands preserves the machine configuration, the machine state, and stores `val` to the address in `rn`.

4.4.14 SUB

The `SUB` instruction takes three register operands. When executing this instruction, the processor reads the terms from the registers `rn` and `rm`, subtracts them, and stores the resulting difference in `rd`.

```
definition
  sub_reg :: "4 word => 4 word => 4 word => MachineCode"
where
  "sub_reg rd rn rm = always (sub_reg1 rd rn rm)"
```

Listing 4.49: The `sub_reg` definition. The definition takes a 4-bit word `rd`, a 4-bit word `rn`, a 4-bit word `rm` and returns the machine code corresponding to an unconditional `SUB` instruction.

We proved the correctness of the `SUB` mnemonic, except for one subgoal (asking us to prove two's complement arithmetic), which we have left as an assumption. We show the resulting lemma in Listing 4.50. We assume that all registers used are general-purpose, and we conclude that after executing this instruction, `rd` contains the difference of the values contained in `rn` and `rm` before executing it.

```
lemma sub_reg_correct:
  "[|Fetch t = (sub_reg rd rn rm, ft);
    general_purpose_reg rd;
    general_purpose_reg rn;
    general_purpose_reg rm;
    machine_config t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        machine_state_preserved t t' &
        REG t' = (REG t)(bin_to_reg rd := REG t (bin_to_reg rn) -
↪  REG t (bin_to_reg rm),
                         RName_PC := REG t RName_PC + 4)"
  sorry
```

Listing 4.50: The `sub_reg_correct` proof. We show that executing an `SUB` instruction with expected operands produces the expected value in `rd`, and preserves the machine configuration and the machine state.

4.5 Compiler

In this section, we describe the resulting compiler, capable of taking the language described in section 2.4 and outputting a list of instructions described in section 4.4. There is one function for compiling each statement and one function for compiling each expression. We present each of these functions in detail, each in conjunction with corresponding proof or assumptions, going from bottom to top. We end the section by showing the primary compiler function, together with the compiler correctness theorem.

4.5.1 Arithmetic expressions

The arithmetic expressions are, as stated in section 3.1, limited to being non-recursive, and hence, the code emitted by the functions compiling these expressions often only consist of one instruction.

In general, all expressions:

- Expect their first operand in register `R0`
- Expect their second operand in register `R1` (if applicable)
- Store their results in register `R0`
- Preserve the machine configuration (described in section 4.1.1)
- Preserve the machine state (described in section 4.1.2)
- Preserve the registers, except for `R0`, `R1`, and `PC`

Also, all arithmetic expressions preserve the flags, but this turned out to be not necessary to prove. It is only meaningful to reason about how an expression behaves if the processor executes it, and hence, we always assume that the `PC` points to the start of each expression when describing the proofs. Using the same argument, we also always let the `PC` point the next basic block after executing an expression. Since all expressions preserve the machine configuration and machine state, we will not discuss that when describing the proofs.

When reasoning about arithmetic expressions, we assume that they evaluate to some value. To express this, we use the function `aval`, defined by Syeda and Klein [10]. This function evaluates the expression following the semantics of the source language, and the goal of all proofs is to prove that the instructions produced when compiling arithmetic expressions produce the same result as `aval`.

4.5.1.1 Loading values to registers

The source language defines values as 32 bits, while the `MOV (immediate)` instruction only allows for values that are 8 bits long. Hence, to load all possible values into a register, there is a need for an alternative solution. Our solution to the problem, shown in Listing 4.51, takes a register and a value as defined by the source language. In case the value is less than 256 (i.e., the 24 most significant bits are zero), the compiler emits a `MOV` instruction. Otherwise, the compiler emits a `B` instruction, the value itself, and then a `LDR (literal)` instruction. The branch jumps over the next instruction (the value), and the `LDR` instruction loads the value into the register.

```
definition
  comp_aexp_mov :: "4 word => 32 word => MachineCode list"
where
  "comp_aexp_mov rt v = (
    if word_extract 31 8 v = (0::24 word)
    then [mov_imm rt (ucast v)]
    else [b_imm 0, (ARM v), ldr_lit 0 rt 12]
  )"
```

Listing 4.51: The `comp_aexp_mov` definition, taking a 4-bit word `rt`, a 32-bit word `v`, and returning a list of instructions in the form of machine codes. If v is less than 256, the definition returns a list consisting of a single `MOV` instruction. Otherwise, the definition returns a list consisting of a series of instructions with `v` embedded as an independent word. In both cases, the returned list of instructions loads `v` into `rt` when executed.

While we do not prove the correctness of loading all possible values (due to using the `LDR (literal)` instruction), we show our assumption in Listing 4.52. We assume that if `reg` is a general purpose register, and there is some value `val`, then there exist `k` steps such that `reg` will have value `val`.

```
lemma comp_aexp_mov_correct:
  "[|code_installed t (comp_aexp_mov reg val);
    general_purpose_reg reg;
    machine_config t;
    state_rel s t|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel s t' &
        REG t' = (REG t)(bin_to_reg reg := val,
                         RName_PC := REG t RName_PC + 4 *
↪   (word_of_int (int (length (comp_aexp_mov reg val)))))"
```

Listing 4.52: The `comp_aexp_mov_correct` proof. We show that executing the instructions returned by `comp_aexp_mov reg val` produces the expected value in `reg`, and preserves the machine configuration and the state relationship.

4.5.1.2 Unary arithmetic expressions

The implementation of unary arithmetic expressions (shown in listing 4.53) is straightforward since we only define negation in the language. As mentioned, the expression expects the presence of the input operand in `R0`, so `NEG` (described in 4.4.12) is the only instruction emitted by the compiler.

```
fun
  comp_aunop :: "aunop => MachineCode list"
where
  "comp_aunop Neg = [neg 0 0]"
```

Listing 4.53: The `comp_aunop` function, taking a `aunop` and returning a list of instructions in the form of machine codes. There is only a single case, and the function returns a list consisting of a single `NEG` instruction. The instruction negates the value in `R0` and stores the result in `R0`.

The proof of unary arithmetic expressions is also straightforward, and we show it in Listing 4.54. We conclude that if there is a unary expression `e` which evaluates to some value `y`, then there exist `k` steps that will result in `R0` having value `y`.

```
lemma comp_aexp_UnOp_correct:
  "[|c = comp_aexp e;
    e = UnOp op x;
    aval e s = Some y;
    code_installed t c;
    state_rel s t;
    machine_config t|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel s t' &
        REG t' = (REG t)(RName_0usr := y,
                         RName_PC := REG t RName_PC + 4 *
↪  (word_of_int (int (length c))))"
```

Listing 4.54: The `comp_aexp_aunop_correct` proof. We show that executing the instructions returned by `comp_aexp (UnOp op x)` in some state `s` produces the expected value `y` in `R0`, and preserves the machine configuration and the state relationship.

4.5.1.3 Binary arithmetic expressions

The implementation of binary arithmetic expressions is analogous to unary arithmetic expressions, except for having two operations and expecting their arguments in `R0` and `R1`. We show it in listing 4.55.

```
fun
  comp_abinop :: "abinop => MachineCode list"
where
  "comp_abinop Plus = [add_reg 0 0 1]" |
  "comp_abinop Minus = [sub_reg 0 0 1]"
```

Listing 4.55: The `comp_abinop` function, taking an `abinop` and returning a list of instructions in the form of machine codes. If the argument equals `Plus`, the function returns a list consisting of a single `ADD` instruction. If the argument equals `Minus`, the function returns a list consisting of a single `SUB` instruction. In both cases, the returned list of instructions performs the expected arithmetic operation using the values in `R0` and `R1`, and stores the result in `R0`.

The proof of binary arithmetic expressions is also analogous, as shown in Listing 4.56. We conclude that if there is some binary arithmetic expression `e` which takes two values `x` and `y`, and evaluates to some value `z`, then there exist `k` steps (one in this case) that will result in `R0` having value `z`. We also conclude that `R1` has value `y`, but this is just an side-effect which we need to take into account.

```
lemma comp_aexp_BinOp_correct:
  "[|aval e s = Some z;
    code_installed t c;
    state_rel s t;
    c = comp_aexp e;
    e = BinOp op x y;
    machine_config t|] ==>
    EX k t'. steps t k = t' &
      machine_config t' &
      state_rel s t' &
      REG t' = (REG t)(RName_0usr := z,
                       RName_1usr := y,
                       RName_PC := REG t RName_PC + 4 * (word_of_int
↪  (int (length c))))"
```

Listing 4.56: The `comp_aexp_BinOp_correct` proof. We show that executing the instructions returned by `comp_aexp (BinOp op x y)` in some state `s` produces the expected value `z` in `R0`, and preserves the machine configuration and the state relationship.

4.5.1.4 Heap lookup expressions

The implementation of heap lookups is simple, considering the definitions presented so far. Since there will be no extensions (such as adding new operations) heap lookup, we decided to put the implementation in the primary arithmetic expression definition, shown in listing 4.58. When the compiler encounters a heap lookup, it emits instructions that load the address into `R0` and then emits an `LDR` instruction,

which overwrites the address in R0 with the value stored in memory.

While we do not prove the heap lookups, we show our assumption in Listing 4.57. We conclude that if there is a heap lookup expression e which evaluates to some value val, then there exist k steps that will result in R0 having value val.

```
lemma comp_aexp_HeapLookup_correct:
  "[|aval e s = Some val;
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_aexp e;
    e = HeapLookup x4|] ==>
    EX k t'. steps t k = t' &
      state_rel s t' &
      machine_config t' &
      REG t' = (REG t)(RName_0usr := val,
                       RName_PC := REG t RName_PC + 4 * (word_of_int
↪   (int (length c))))"
```

Listing 4.57: The comp_aexp_HeapLookup_correct assumption. We show that executing the instructions returned by comp_aexp (HeapLookup a) in some state s produces the expected value val in R0, and preserves the machine configuration and the state relationship.

4.5.1.5 Main arithmetic expression function

With the definitions presented in this subsection at hand, we give the main arithmetic expression function in listing 4.58. We handle different types of expression on a case basis, using the definitions and functions presented above.

```
fun
  comp_aexp :: "aexp => MachineCode list"
where
  "comp_aexp (Const v) = comp_aexp_mov 0 v" |
  "comp_aexp (UnOp op a) = comp_aexp_mov 0 a @ comp_aunop op" |
  "comp_aexp (BinOp op a1 a2) = comp_aexp_mov 0 a1 @ comp_aexp_mov 1
   ↪  a2 @ comp_abinop op" |
  "comp_aexp (HeapLookup a) = comp_aexp_mov 0 a @ [ldr_imm 0 0 0]"
```

Listing 4.58: The `comp_aexp` definition, taking an `aexp` and returning a list of instructions in the form of machine codes. Except for `HeapLookup`, the function uses the definitions and functions described in this subsection to create a list of instructions for each expression type. If the expression is of type `HeapLookup`, the function uses `comp_aexp_mov` in combination with an `LDR (immediate)` instruction. In all cases, the returned list of instructions places the value the arithmetic expression evaluates to in `R0`.

The proof of arithmetic expressions relies upon the lemmas presented in this section, and we give it in listing 4.59. Since there are assumptions among these lemmas, we also assume the correctness of arithmetic expressions. We conclude that if there is some arithmetic expression `e` that produces some value `val`, then there exist `k` steps such that executing them results in `R0` having value `val` and `R1` being undefined (because of the side-effect described for binary expressions).

```
lemma comp_aexp_correct:
  "[|aval e s = Some val;
    code_installed t (comp_aexp e);
    machine_config t;
    state_rel s t|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel s t' &
        REG t' = (REG t)(RName_0usr := val,
                         RName_1usr := (REG t') RName_1usr,
                         RName_PC := REG t RName_PC + 4 *
 ↪  (word_of_int (int (length (comp_aexp e)))))"
```

Listing 4.59: The `comp_aexp_correct` proof. We show that executing the instructions returned by `comp_aexp e` in some state `s` produces the expected value `val` in `R0`, and preserves the machine configuration and the state relationship.

4.5.2 Boolean expressions

In general, boolean expressions follow the same general pattern described for arithmetic expressions in subsection 4.5.1. The main differences are that boolean expressions only expect values zero or one (which we assume in the proofs) as input and

do not preserve the flags.

When reasoning about boolean expressions, we assume that they evaluate to some value. To express this, we use the function `bval`, defined by Syeda and Klein [10]. This function evaluates the expression following the semantics of the source language, and the goal of all proofs is to prove that the instructions produced when compiling boolean expressions produce the same result as `bval`.

4.5.2.1 Loading values to registers

Since boolean expressions only handle values of zero and one, the implementation for moving a constant into a register is analogous to moving a value that is less than 256 into a register for arithmetic expressions. In Listing 4.60, we show our implementation.

```
definition
  comp_bexp_mov :: "4 word => bool => MachineCode list"
where
  "comp_bexp_mov rt v = [mov_imm rt (if v then 1 else 0)]"
```

Listing 4.60: The `comp_bexp_mov` definition, taking a 4-bit word `rt`, a `bool v`, and returning a list of instructions in the form of a list consisting of a single `MOV` instruction. If `v` is true, the returned list of instructions loads one into `rt` when executed. Otherwise, it loads zero into `rt`.

Listing 4.61 shows our proof for loading boolean values. We conclude that if `reg` is a general-purpose register, and there is some boolean value `val`, then there exist `k` steps such that the register will have the value one in case `val` is true, and zero otherwise.

```
lemma comp_bexp_mov_correct:
  "[|code_installed t (comp_bexp_mov reg val);
    general_purpose_reg reg;
    machine_config t;
    state_rel s t|] ==>
      EX t'. steps t 1 = t' &
        machine_config t' &
        state_rel s t' &
        REG t' = (REG t)(bin_to_reg reg := (if val then 1 else 0),
                         RName_PC := REG t RName_PC + 4 *
↪  (word_of_int (int (length (comp_bexp_mov reg val)))))"
```

Listing 4.61: The `comp_bexp_mov_correct` proof. We show that executing the instructions returned by `comp_bexp_mov reg val` produces the expected value in `reg`, and preserves the machine configuration and the state relationship.

4.5.2.2 Unary boolean expressions

The implementation of unary boolean expressions (shown in Listing 4.62) consists of one case since we only define logical not in the source language. The compiler emits a `CMP` instruction, which compares the value in `R0` with zero. Then, the compiler emits two conditional `MOV` instructions, setting `R0` to one in case of the `Z` flag being set, and zero otherwise, effectively negating the boolean value present in `R0` before executing the expression.

```
fun
  comp_bunop :: "bunop => MachineCode list"
where
  "comp_bunop Not = [cmp_imm 0 0, moveq_imm 0 1, movne_imm 0 0]"
```

Listing 4.62: The `comp_bunop` definition, taking a `bunop` and returning a list of instructions in the form of machine codes. There is only a single case, and the function returns a list consisting a `CMP` and two conditional `MOV` instructions. The instructions perform a logical not on the value in `R0` and stores the result in `R0`.

The proof of unary boolean expressions is straightforward, and we show it in Listing 4.63. We conclude that if there is a unary expression `b` which evaluates to some boolean value `val`, then there exist `k` steps that will result in `R0` having value one in case `val` is true, and zero otherwise.

```
lemma comp_bexp_BUnOp_correct:
  "[|bval b s = Some val;
    code_installed t c;
    machine_config t;
    state_rel s t;
    b = BUnOp op a;
    c = comp_bexp b|] ==>
    EX k t'. steps t k = t' &
      machine_config t' &
      state_rel s t' &
      REG t' = (REG t)(RName_0usr := (if val then 1 else 0),
                       RName_PC := REG t RName_PC + 4 * (word_of_int
↪  (int (length c))))"
```

Listing 4.63: The `comp_bexp_BUnOp_correct` proof. We show that executing the instructions returned by `comp_bexp (BUnOp op a)` in some state `s` produces the expected value in `R0`, and preserves the machine configuration and the state relationship.

4.5.2.3 Binary boolean expressions

We show the implementation of binary boolean expressions in Listing 4.64, and it is analogous to binary arithmetic expressions, except for only storing zero or one in

R0.

```
fun
  comp_bbinop :: "bbinop => MachineCode list"
where
  "comp_bbinop And = [and_reg 0 0 1]" |
  "comp_bbinop Or = [or_reg 0 0 1]"
```

Listing 4.64: The `comp_bbinop` definition, taking an `bbinop` and returning a list of instructions in the form of machine codes. If the argument equals `And`, the function returns a list consisting of a single `AND` instruction. If the argument equals `Or`, the function returns a list consisting of a single `OR` instruction. In both cases, the returned list of instructions performs the expected boolean operation using the values in `R0` and `R1`, and stores the result in `R0`.

The proof for binary boolean expressions (shown in Listing 4.65) is also analogous with the proof for binary arithmetic expressions, except for only storing zero or one in `R0`.

```
lemma comp_bexp_BBinOp_correct:
  "[|bval b s = Some z;
    code_installed t c;
    machine_config t;
    state_rel s t;
    b = BBinOp op x y;
    c = comp_bexp b|] ==>
    EX k t'. steps t k = t' &
      machine_config t' &
      state_rel s t' &
      REG t' = (REG t)(RName_0usr := (if z then 1 else 0),
                       RName_1usr := REG t' RName_1usr,
                       RName_PC := REG t RName_PC + 4 * (word_of_int
↪  (int (length c))))"
```

Listing 4.65: The `comp_bexp_BBinOp_correct` proof. We show that executing the instructions returned by `comp_bexp (BBinOp op x y)` in some state `s` produces the expected value in `R0`, and preserves the machine configuration and the state relationship.

4.5.2.4 Comparison expressions

The implementation of comparison expressions (shown in Listing 4.66) consists of one case since we only define less than in the source language. The compiler emits a `CMP` instruction, which compares the value in `R0` with the value in `R1`. Then, the compiler emits two conditional `MOV` instructions, setting `R0` to one in case the value in `R0` is less than the value in `R1`, and zero otherwise.

```
fun
  comp_bcomp :: "bcomp => MachineCode list"
where
  "comp_bcomp Less = [cmp_reg 0 1, movlt_imm 0 1, movge_imm 0 0]"
```

Listing 4.66: The `comp_bcomp` definition, taking a `bcomp` and returning a list of instructions in the form of machine codes. There is only a single case, and the function returns a list consisting a `CMP` and two conditional `MOV` instructions. The instructions perform a comparison of the values in `R0` and `R1`, and store one in `R0` if the value in `R1` is less than the value in `R0`. Otherwise, they store zero in `R0`.

While we do not prove comparisons (due to the uncertainty of how to handle errors in the arithmetic subexpressions), we show our assumption in Listing 4.67. We conclude that if there is a comparison expression `e` which evaluates to some value `val`, then there exist `k` steps that will result in `R0` having value one in case `val` is true, and zero otherwise.

```
lemma comp_bexp_BComp_correct:
  "[|bval b s = Some val;
    code_installed t c;
    machine_config t;
    state_rel s t;
    b = BComp op a1 a2;
    c = comp_bexp b|] ==>
    EX k t'. steps t k = t' &
      machine_config t' &
      state_rel s t' &
      REG t' = (REG t)(RName_0usr := (if val then 1 else 0),
                       RName_1usr := REG t' RName_1usr,
                       RName_PC := REG t RName_PC + 4 * (word_of_int
↪  (int (length c))))"
```

Listing 4.67: The `comp_bexp_BComp_correct` proof. We show that executing the instructions returned by `comp_bexp (BComp op a1 a2)` in some state `s` produces the expected value in `R0`, and preserves the machine configuration and the state relationship.

4.5.2.5 Main boolean expression function

With the definitions presented in this subsection at hand, we give the primary boolean expression function in Listing 4.68. We handle different types of expression on a case basis, using the definitions and functions presented above.

```
fun
  comp_bexp :: "bexp => MachineCode list"
where
  "comp_bexp (BConst v) = comp_bexp_mov 0 v" |
  "comp_bexp (BUnOp op b) = comp_bexp_mov 0 b @ comp_bunop op" |
  "comp_bexp (BBinOp op b1 b2) = comp_bexp_mov 0 b1 @ comp_bexp_mov
   ↪  1 b2 @ comp_bbinop op" |
  "comp_bexp (BComp op a1 a2) = comp_aexp a1 @ mov_reg 0 2 #
   ↪  comp_aexp a2 @ mov_reg 2 1 # comp_bcomp op"
```

Listing 4.68: The `comp_bexp` definition, taking an `bexp` and returning a list of instructions in the form of machine codes. Except for `BComp`, the function uses the definitions and functions described in this subsection to create a list of instructions for each expression type. If the expression is of type `BComp`, the function uses `comp_aexp` in combination with `MOV (reg)` instructions. The reason for using the `MOV` instructions is to safely stash away the value produced by the first arithmetic expression in `R2` while the second arithmetic expression executes. In all cases, the returned list of instructions places the value the boolean expression evaluates to in `R0`.

We show our correctness assumption for boolean expressions in Listing 4.69. We conclude that if there is some boolean expression `b` that produces some boolean value `val`, then there exist `k` steps such that executing them results in `R0` having value `val` and `R1` being undefined.

```
lemma comp_bexp_correct:
  "[|bval e s = Some val;
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_bexp e|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel s t' &
        REG t' = (REG t)(RName_0usr := (if val then 1 else 0),
                         RName_1usr := REG t' RName_1usr,
                         RName_PC := REG t RName_PC + 4 *
 ↪  (word_of_int (int (length c))))"
```

Listing 4.69: The `comp_bexp_correct` proof. We show that executing the instructions returned by `comp_bexp e` in some state `s` produces the expected value `val` in `R0`, and preserves the machine configuration and the state relationship.

4.5.3 Statements

This subsection describes how we implement and reason about statements in the compiler. As with expressions, in each proof, we always assume that the `PC` points to the start of the statement, and that it points to the next statement in the conclusion. Another important property shared between all statements is that they preserve the machine configuration and the state relationship. This is why `machine_config` and `state_rel` occur both in the assumptions and in the conclusion for each statement.

When reasoning about statements, we assume that they execute without any error. To express this, we use the operand `=>`, defined by Syeda and Klein [10] in our statement proofs and assumptions. This operand executes the statement in accordance with the semantics of the source language, and the goal of all proofs is to prove that the instructions produced when compiling statements have the same effect as executing the statement using `=>`.

4.5.3.1 Assign statements

The implementation of assign statements is part of the primary compiler function, shown in listing 4.78. The compiler begins by emitting instructions for the first arithmetic expression, placing the address in `R0`. Because the upcoming expression will overwrite `R0` and potentially `R1`, the compiler then emits a `MOV` instruction, moving the address to `R2`. The compiler then emits instructions for the second arithmetic expression, placing the value in `R0`. Finally, the compiler emits a `STR` instruction, storing the value in `R0` to the address in `R2`.

We show our correctness assumption for assign statements in listing 4.70. We assume there is an arithmetic expression `lval` that evaluates to some writable address `vp` and an arithmetic expression `rval` that evaluates to some value `v`. Then, we conclude that there is `k` such that executing `k` steps will result in address `vp` having value `v`, and registers `R0`, `R1`, and `R2` being undefined.

```
lemma comp_Assign_correct:
  "[|aval lval s = Some vp;
    aval rval s = Some v;
    Addr vp ~: incon_set s;
    addr_trans s (Addr vp) = Some pp;
    aligned pp;
    c = comp_com (Assign lval rval);
    code_installed t c;
    machine_config t;
    Some (s(|heap := heap s(pp |-> v), incon_set := iset_upd s pp v,
↪   p_state.global_set := gset_upd s pp v|)) ~= None;
    state_rel s t|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some (s(|heap := heap s(pp |-> v), incon_set
↪   := iset_upd s pp |, p_state.global_set := gset_upd s pp v|))))
↪   t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
    sorry
```

Listing 4.70: The `comp_Assign_correct` assumption. We show that executing the instructions returned by `comp_com (Assign lval rval)` in some state `s` produces the expected value `val` at address `vp`, and preserves the machine configuration and the rest of the state relationship.

4.5.3.2 Sequence statements

Sequence statements are, due to their simplicity, also part of the primary compiler function. When encountering a sequence statement, the compiler first emits instructions for the first statement, then the second.

We show the correctness assumption for sequence statements in Listing A.3.

We assume that if there exists a state `s1` and a statement `p1`, then there exist `ka` steps such that executing them results in state `s2` and registers `R0`, `R1`, and `R2` being undefined. Since the steps results in a new program state, this implies that `p1` terminates.

Further, we assume that if there exists a state `s2` (the same as in our first assumption) and a statement `p2`, then there exist `kb` steps such that executing them results in state `y` and registers `R0`, `R1`, and `R2` being undefined. With the same reasoning as in the first assumption, `p2` also terminates.

With these assumptions, we conclude that there is k (which is ka + kb) such that executing k steps results in state y and registers R0, R1, and R2 being undefined.

4.5.3.3 If statements

We give our implementation of if statements in listing 4.78. The compiler begins by compiling the statement c1 into some instructions i1 and c2 into some instructions i2. The compiler will, however, defer emitting the instructions.

The compiler then emits

- Instructions for the boolean expression b
- A CMP instruction, which sets the Z flag in case b evaluates to false
- A BEQ instruction, which jumps over i1 and the upcoming B instruction in case the Z flag is set
- The instructions in i1
- A B instruction, which jumps over i2
- The instructions in i2

In combination, these instructions executes c1 in case the boolean expression evaluates to true, and c2 otherwise.

The correctness assumption consists of two cases: One is when the boolean expression b evaluates to true and the other when it evaluates to false.

We begin by showing the case when b evaluates to true in Listing A.1.

We assume that if there exists a state s and a statement p1, then there exist ka steps such that executing them results in state y and registers R0, R1, and R2 being undefined. We conclude that there is k such that executing k steps results in state y and registers R0, R1, and R2 being undefined.

We continue with the case when b evaluates to false in listing A.2.

We assume that if there exists a state s and a statement p2, then there exist kb steps such that executing them results in state y and registers R0, R1, and R2 being undefined. We conclude that there is k such that executing k steps results in state y and registers R0, R1, and R2 being undefined.

When combined, these two cases constitute our correctness assumption for if statements.

4.5.3.4 While statements

We give our implementation of while statements in listing 4.78. The compiler begins by compiling the boolean expression b into some instructions i1 and the statement c into some instructions i2.

The compiler then emits

- The instructions in `i1`
- A `CMP` instruction, which sets the `Z` flag in case `b` evaluates to false
- A `BEQ` instruction, which jumps over `i2` and the upcoming `B` instruction in case the `Z` flag is set
- The instructions in `i2`
- A `B` instruction, which jumps back to the beginning of `i1`

In combination, these instructions executes `c` while `b` evaluates to true, and branches out of the loop otherwise.

The correctness assumption consists of two cases: One is when the boolean expression `b` evaluates to true and the other when it evaluates to false.

We begin by showing the case when `b` evaluates to false in Listing 4.71. Since `c` does not execute in this case, we conclude that there exists k such that executing k steps (which include the boolean expression) retain some state `y`.

```
lemma comp_WhileFalse_correct:
  "[|bval b y = Some False;
    code_installed t c;
    machine_config t;
    state_rel y t;
    c = comp_com (While b ca)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some y)) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
  sorry
```

Listing 4.71: The `comp_WhileFalse_correct` assumption. We show that executing the instructions returned by `comp_com (While b ca)` in some state `s` produces some new state `y`, and preserves the machine configuration and the state relationship.

We continue with the case when `b` evaluates to true in listing A.4.

We assume that if there exists a state `s` and a statement `p1`, then there exist `ka` steps such that executing them results in state `s1` and registers `R0`, `R1`, and `R2` being undefined. Also, we assume that if the given while loop executes in state `s1`, then there exist `kb` steps such that executing them results in state `y` and registers `R0`, `R1`,

and R2 being undefined. We then conclude that there is k (which is ka + kb) such that executing k steps results in state y and registers R0, R1, and R2 being undefined.

When combined, these two cases constitute our correctness assumption for while statements.

4.5.3.5 Flush statements

The source language defines four types of flush statements, and we show our implementation of them in Listing 4.72.

The first case is the flushTLB type, which invalidates all entries in the TLB. The only thing the compiler needs to emit is a TLBIALL instruction, described in subsection 4.4.7.

The second case is the flushASID type, which invalidates all entries in the TLB belonging to a given ASID, which is a process in the ARM architecture. The compiler emits a MOV instruction, which stores the ASID in R0. Despite the MOV instructions only being capable of handling 8 bits, that is sufficient since ASID:s are also 8 bits long. Then, the compiler emits a TLBIASID instruction with its operand set to R0, described in subsection 4.4.7.

The third case is the flushvarange, which invalidates all entries in the TLB belonging to a given address. The compiler emits instructions for loading the address into R0, using the same pattern as for loading large values as described in subsection 4.5.1. Finally, the compiler issues a TLBIMVAA instruction with its operand set to R0, described in subsection 4.4.7.

The last case is the flushASIDvarange, which invalidates all entries in the TLB belonging to a given ASID and a given address. The compiler emits instructions for loading a data structure consisting of the ASID and the address into R0, using the same pattern as for loading large values as described in subsection 4.5.1. Finally, the compiler issues a TLBIMVA instruction with its operand set to R0, described in subsection 4.4.7.

```
fun
  comp_flush :: "MMU_Prg_Logic.flush_type => MachineCode list"
where
  "comp_flush flushTLB = [tlbiall]" |
  "comp_flush (flushASID a) = [mov_imm 0 (ucast a), tlbiasid 0]" |
  "comp_flush (flushvarange va) = [
    b_imm 0,
    ARM (addr_val va),
    ldr_lit 0 0 12,
    tlbimvaa 0
  ]" |
  "comp_flush (flushASIDvarange a va) = [
    b_imm 0,
    ARM (word_cat a (word_extract 23 0 (addr_val va)::24 word)),
    ldr_lit 0 0 12,
    tlbimva 0
  ]"
```

Listing 4.72: The `comp_flush` definition, taking a `flush_type` and returning a list of instructions in the form of machine codes. The function uses the definitions, functions, and patterns described in subsection 4.4.7 and 4.5.1 to create a list of instructions for each flush type. If the flush is of type `flushASIDvarange`, the function uses `word_cat` and `word_extract` to construct the argument required by `TLBIVMA`. In all cases, the returned list of instructions flushes the entries in the TLB specified by the source language.

While not proving the flush statement, we show our correctness assumption in Listing 4.73. We conclude that if there is some flush statement of type `f`, then executing it in some state `t` produces a state `t'` such that the TLB is updated in accordance with the semantics of the source language, and with `R0`, `R1`, and `R2` being undefined.

```
lemma comp_Flush_correct:
  "[|mode s = Kernel;
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_com (Flush f)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some (s(|incon_set := flush_effect_iset f
↪   (incon_set s) (p_state.global_set s) (asid s),
                                p_state.global_set :=
↪   flush_effect_glb f (p_state.global_set s) (asid s) (heap s)
↪   (root s),
                                ptable_snapshot := flush_effect_snp
↪   f (ptable_snapshot s) (asid s)|)))) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
```

Listing 4.73: The `comp_Flush_correct` assumption. We show that executing the instructions returned by `comp_com (Flush f)` in some state `s` produces some new state with TLB entries invalidated in accordance with the source language semantics, and preserves the machine configuration and the state relationship.

4.5.3.6 Update TTBR0 statements

The source language defines a statement for changing the page table root called update TTBR0. We implement this statement in the primary compiler function, shown in listing 4.78.

The compiler begins by emitting instructions for the arithmetic expression `rte`, placing the page table root address in `R0`. Then, the compiler emits the `SETTTBR0` instruction, defined in subsection 4.4.7.

While not proving the update TTBR0 statement, we show our correctness assumption in Listing 4.74. We conclude that if there is some update TTBR0 statement, then executing it in some state `t` produces a state `t'` such that the page table root is set to `rte`, the TLB is updated in accordance with the semantics of the source language, and with `R0`, `R1`, and `R2` being undefined.

```
lemma comp_UpdateTTBR0_correct:
  "[|mode s = Kernel;
    aval rte s = Some rt;
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_com (UpdateTTBR0 rte)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some (s(|root := Addr rt, incon_set :=
↪   iset_upd' s rt, p_state.global_set := gset_upd' s rt|)))) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
  sorry
```

Listing 4.74: The `comp_UpdateTTBR0_correct` assumption. We show that executing the instructions returned by `comp_com (UpdateTTBR0 rte)` in some state `s` produces some new state with `rt` as page table root, with TLB entries invalidated in accordance with the source language semantics, and preserves the machine configuration and the state relationship.

4.5.3.7 Update ASID statements

The source language defines a statement for changing the current process called update ASID. We implement this statement in the primary compiler function, shown in listing 4.78.

We show our correctness assumption in listing 4.75. We conclude that if there is some update ASID statement, then executing it in some state `t` produces a state `t'` such that the process is set to `v`, the TLB is updated in accordance with the semantics of the source language, and with `R0`, `R1`, and `R2` being undefined.

```
lemma comp_UpdateASID_correct:
  "[|mode s = Kernel;
    c = comp_com (UpdateASID a);
    code_installed t c;
    machine_config t;
    state_rel s t|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some (s(|asid := a,
                                 incon_set := incon_load (cur_pt_snp'
↪   (ptable_snapshot s) (incon_set s) (heap s) (root s) (asid s))
↪   (incon_set s) (p_state.global_set s) a (heap s) (root s),
                                 ptable_snapshot := cur_pt_snp'
↪   (ptable_snapshot s) (incon_set s) (heap s) (root s) (asid
↪   s)|)))) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
  sorry
```

Listing 4.75: The `comp_UpdateASID_correct` assumption. We show that executing the instructions returned by `comp_com (UpdateASID a)` in some state `s` produces some new state with `a` as current process, with TLB entries invalidated in accordance with the source language semantics, and preserves the machine configuration and the state relationship.

4.5.3.8 Set mode statements

We split our implementation of set mode into two cases, which we show in Listing 4.76. When researching how to change modes on the ARM architecture, we discovered that the processor (except for using software interrupts) can't switch mode when running in user mode. Consequently, the compiler assumes that the processor is running in kernel mode when executing this statement. Hence, there is no need to emit any instructions for switching to kernel mode. In the case of switching from kernel to user mode, the compiler emits a `MOV` instruction that loads the correct mode into R0 and then emits an `MSR` instruction (described in subsection 4.4.10) that does the actual switch.

```
fun
  comp_set_mode :: "mode_t => MachineCode list"
where
  "comp_set_mode Kernel = []" |
  "comp_set_mode User = [mov_imm 0 0x10, msr_reg 0 0x1 0]"
```

Listing 4.76: The `comp_set_mode` definition, taking a `mode_t`, and returning a list of instructions in the form of machine codes. The compiler assumes that these instructions run in kernel mode, so if the new mode is `Kernel`, the function returns an empty list. If the new mode is `User`, the function returns a list of instructions consisting of a `MOV` and an `MSR` instruction, described in section 4.4. In all cases, the returned list of instructions makes the processor run in the requested mode after executing the instructions returned by this function.

We show our proof assumption for set mode in Listing 4.77. We conclude that if the processor runs in kernel mode, then there is `k` steps such that after executing them, the processor runs in mode `m`.

```
lemma comp_SetMode_correct:
  "[|mode s = Kernel;
    code_installed t (comp_com (SetMode m));
    machine_config t;
    state_rel s t;
    c = comp_com (SetMode m)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some (s(|mode := m|)))) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪  word_of_int (int (length (comp_com (SetMode m)))))"
```

Listing 4.77: The `comp_SetMode_correct` assumption. We show that executing the instructions returned by `comp_com (SetMode m)` in some state `s` produces some new state with `m` as mode, and preserves the machine configuration and the state relationship.

4.5.3.9 Main compiler function

Having the definitions presented in this section, we give the main compiler function in listing 4.78. We handle each statement defined by the source language of a case basis, as described above.

```
fun
  comp_com :: "com => MachineCode list"
where
  "comp_com SKIP = []" |
  "comp_com (Assign addr val) = comp_aexp addr @ mov_reg 2 0 #
   ↪  comp_aexp val @ [str_imm 0 2 0]" |
  "comp_com (Seq c1 c2) = (comp_com c1) @ (comp_com c2)" |
  "comp_com (If b c1 c2) = (
    let i1 = comp_com c1;
        i2 = comp_com c2
    in (
      comp_bexp b @
      cmp_imm 0 0 #
      beq_imm ((code_size i1)-1) #
      i1 @ b_imm ((code_size i2)-1) #
      i2
    )
  )" |
  "comp_com (While b c) = (
    let i1 = comp_bexp b;
        i2 = comp_com c
    in (
      i1 @
      cmp_imm 0 0 #
      beq_imm ((code_size i2)-1) #
      i2 @
      [b_imm (-((code_size i1) + (code_size i2) + 4))]
    )
  )" |
  "comp_com (Flush t) = comp_flush t" |
  "comp_com (UpdateTTBR0 a) = comp_aexp a @ [setttbr0 0]" |
  "comp_com (UpdateASID v) = [mov_imm 0 (ucast v), setasid 0]" |
  "comp_com (SetMode m) = comp_set_mode m"
```

Listing 4.78: The `comp_com` function, taking a `com` and returning a list of instructions in the form of machine codes. The function uses the definitions, functions, and patterns described in this section and section 4.4 to create a list of instructions for each statement. If the statement is of type `Seq`, `If`, or `While` the function uses itself, making it recursive. In all cases, the returned list of instructions has the same semantics as the source language defined by Syeda and Klein.

With the definitions and lemmas presented above, we arrive at the compiler correctness theorem, shown in Listing 4.79. We prove it by induction and conclude that if there is a statement `p` and a state `t`, then there is `k` such that taking `k` steps will result in a new state `t'` which differs from `t` in accordance with the semantics of the WHILE language.

```
theorem comp_com_correct:
  "[|(p,s) => st;
    c = comp_com p;
    code_installed t c;
    machine_config t;
    st ~= None;
    state_rel s t|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the st) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                  RName_1usr := REG t' RName_1usr,
                  RName_2usr := REG t' RName_2usr,
                  RName_PC := REG t RName_PC + 4 * (word_of_int (int
↪   (length c))))"
```

Listing 4.79: The `comp_com_correct` proof. We show that executing the instructions returned by `comp_com p` in some state `s` preserves the machine configuration and the state relationship.

5

Discussion

Having implemented the compiler and stated the compiler correctness theorem, we continue by discussing the roadblocks encountered, and our suggested solutions to the remaining problems.

Before improving anything else, we propose revisiting each layer and separating them more strictly. We feel that during the project, we have been helped a lot by the layered design, but there is still room for improvement. When working with a top-down approach, items tend to trickle down into layers where they do not belong. For example, we have experienced the state as defined by the source language (belonging firmly to the compiler layer) trickle down into the assembly layer during the project, and some of these errors remain. We also experienced another problem related to top-down design. When we drafted our first version of the compiler correctness theorem, we made a small mistake (we assumed that taking `k` steps always meant incrementing `PC` with `4k`, something that is obviously wrong when the language has statements such as conditional execution and loops) which cost us several days of work. Even though we were able to salvage much progress despite the error, it still made us question our choice of using a top-down approach. In hindsight, however, it would have been hard to determine which constructs to use without a top-down approach, and therefore, we consider using it the right choice.

As we see it, the next logical step is to evaluate the changes made to the source language, especially by reverting the change not to allow recursive expressions. Another change that seems apparent to us is to reconsider our choices of supported arithmetic and boolean operations, which would mean adding more operations. After all, there is nothing wrong with the concrete operations we chose to implement. Given the compiler and assembler's current state, we do not estimate that implementing such changes to the compiler would require much work. However, making changes to the compiler would require a partial reevaluation of the compiler correctness theorem, which we estimate to be a significantly larger workload. Therefore, in case such changes are to be made, we recommend making them before attempting to prove the compiler correct.

After evaluating the source language and compiler, we suggest proving the simple lemmas (such as two-complement being true, as described in subsection 4.4.12) left as assumptions by us. Proving these lemmas implies proving most instructions, and we therefore also suggest generalizing the instruction proofs. We want to point out that the proofs for some instructions assume the operands have specific values,

which is a by-product of using a top-down approach. While not crucial for proving the compiler correctness theorem, generalizing the instruction proofs would make it easier to make modifications (such as extending the present features) to the compiler.

During this project, we were unable to prove the correctness of heap lookups due to alignment issues. We should consider relaxing the assumptions that the processor uses aligned addresses, since (assuming the reversion of the restrictions on recursive expressions) the addresses used by heap lookup are not guaranteed to be aligned. A similar occurred when we tried to prove the assignment of memory, described in subsection 4.5.3.1.

When we look at the issues described in this section, memory accesses are a common denominator. Our interpretation of this fact is that it is necessary to review our machine configuration assumptions and our lemmas dealing with memory access in the compiler layer. To prove the memory accesses, it is therefore probably necessary to derive lemmas stating that if an address is accessible at the compiler layer, then it is also accessible at the assembler layer. Also, we suggest adding additional lemmas for reasoning about code installed in memory. As noted in subsection 4.1.4 and subsection 4.4.6, we would most likely have been able to prove the correctness of loading large values into registers if we had such lemmas.

6

Conclusion

In this thesis, we have described the implementation of a compiler targeting the ARM architecture for the language described by Syeda and Klein, culminating in the statement of the compiler correctness theorem for the said compiler.

To get to point where we were able to state the correctness theorem, we had to modify the source language and make it less powerful by removing recursive expressions. These modifications might seem like a step backward, but considering the large amount of work we put into reasoning about the modified language, we conclude that it was the right decision. As discussed, the modified language is still useful when reasoning about the TLB, and should someone continue our work; they will probably revert these changes in the future.

As an intermediate step, we modified the Cambridge ISA to support flush, bridging work by Myreen and Fox on the ISA with work by Syeda and Klien on TLB consistency. While we only added flush, we concluded that another benefit of this is that we laid the groundwork for adding more features related to the coprocessors on ARM.

As an intermediate step, we also developed an assembler, enabling the use of mnemonics in the project. While we did not take this step into account at the beginning of the project, we conclude that it is necessary to have an assembler. The alternative would have been to use the Cambridge ISA instructions directly in the compiler, something that would have been more difficult to implement, and less portable. Another benefit not apparent at first glance is that the instruction proofs are reusable, simplifying reasoning about constructs relying upon them (such as the compiler) significantly. The assembler is easy to extend with further mnemonics, and it should be possible to reuse it for other projects in the future.

Finally, we implemented a compiler capable of compiling the source language into machine language. While not proving the implementation in its entirety, we proved it partially. Seen from another angle, we did not find anything indicating that the answer to our research question is negative. This conclusion is further reinforced by the rather simple state relationship, which is a consequence of the source language and the previous work on the TLB being closely related. Given that we or someone else addresses the issues discussed in chapter 5, and given enough time and resources, we firmly believe that it is possible to use our correctness theorem to verify the compiler created during this project.

When writing proofs for the compiler, it was also apparent that conclusions from the compiler layer trickled up to the compiler layer. This observation is important, since it indicates to us that the answer to our second research question is yes. While not being able to investigate if the compiler preserve the TLB consistency of a program, we can at least conclude that it is presumable.

Bibliography

[1] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[2] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1), February 2014.

[3] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[4] L3: A Specification Language for Instruction Set Architectures, `https://acjf3.github.io/l3/index.html` (accessed in June 2020)

[5] Anthony Fox. Directions in ISA specification. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 338–344, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 243–258, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] Hira Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 490–508. EasyChair, 2017.

[8] Isabelle Documentation, `https://isabelle.in.tum.de/documentation.html` (accessed in June 2020)

[9] ARM Ltd. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R*, 2008.

[10] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 542–559, Cham, 2018. Springer International Publishing.

[11] The Coq Proof Assistant, `https://coq.inria.fr/` (accessed in June 2020)

[12] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[13] Tobias Nipkow. *Concrete semantics : with Isabelle/HOL*. Springer, Cham, 2014.

[14] Alfred Aho. *Compilers : principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2007.

[15] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, October 1980.

A

Appendix 1

```
lemma comp_IfTrue_correct:
  "[|bval b s = Some True;
    (p1, s) => Some y;
    !! ta. [|c1 = comp_com p1;
           code_installed ta c1;
           machine_config ta;
           Some y ~= None;
           state_rel s ta|] ==>
            EX ka ta'. steps ta ka = ta' &
              machine_config ta' &
              state_rel (the (Some y)) ta' &
              REG ta' = (REG ta)(RName_0usr := REG ta' RName_0usr,
                                 RName_1usr := REG ta' RName_1usr,
                                 RName_2usr := REG ta' RName_2usr,
                                 RName_PC := state.REG ta RName_PC +
↪   4 * word_of_int (int (length c1)));
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_com (If b p1 p2)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some y)) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
    sorry
```

Listing A.1: The `comp_IfTrue_correct` assumption. Assuming that `p1` executes correctly, we show that executing the instructions returned by `comp_com (If b p1 p2)` in some state `s` produces some new state `y`, and preserves the machine configuration and the state relationship.

```
lemma comp_IfFalse_correct:
  "[|bval b s = Some False;
    (p2, s) => Some y;
    !! tb. [|c2 = comp_com p2;
          code_installed tb c2;
          machine_config tb;
          Some y ~= None;
          state_rel s tb|] ==>
            EX k tb'. steps tb k = tb' &
              machine_config tb' &
              state_rel (the (Some y)) tb' &
              REG tb' = (REG tb)(RName_0usr := REG tb' RName_0usr,
                                 RName_1usr := REG tb' RName_1usr,
                                 RName_2usr := REG tb' RName_2usr,
                                 RName_PC := REG tb RName_PC + 4 *
↪   word_of_int (int (length c2)));
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_com (If b p1 p2)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some y)) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
  sorry
```

Listing A.2: The `comp_IfFalse_correct` assumption. Assuming that `p2` executes correctly, we show that executing the instructions returned by `comp_com (If b p1 p2)` in some state `s` produces some new state `y`, and preserves the machine configuration and the state relationship.

```
lemma comp_Seq_correct:
  "[|(p1, s1) => Some s2;
    !! ta. [|c1 = comp_com p1;
          code_installed ta c1;
          machine_config ta;
          Some s2 ~= None;
          state_rel s1 ta|] ==>
            EX ka ta'. steps ta ka = ta' &
              machine_config ta' &
              state_rel (the (Some s2)) ta' &
              REG ta' = (REG ta)(RName_0usr := REG ta' RName_0usr,
                                 RName_1usr := REG ta' RName_1usr,
                                 RName_2usr := REG ta' RName_2usr,
                                 RName_PC := REG ta RName_PC + 4 *
↪   word_of_int (int (length c1)));
    (p2, s2) => Some y;
    !! tb. [|c2 = comp_com p2;
          code_installed tb c2;
          machine_config tb;
          Some y ~= None;
          state_rel s2 tb|] ==>
            EX kb tb'. steps tb kb = tb' &
              machine_config tb' &
              state_rel (the (Some y)) tb' &
              REG tb' = (REG tb)(RName_0usr := REG tb' RName_0usr,
                                 RName_1usr := REG tb' RName_1usr,
                                 RName_2usr := REG tb' RName_2usr,
                                 RName_PC := REG tb RName_PC + 4 *
↪   word_of_int (int (length c2)));
    code_installed t c;
    machine_config t;
    state_rel s1 t;
    c = comp_com (Seq p1 p2)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some y)) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := state.REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
    sorry
```

Listing A.3: The `comp_Seq_correct` assumption. Assuming that `p1` and `p2` execute correctly, we show that executing the instructions returned by `comp_com (Seq p1 p2)` in some state `s` produces some new state `y`, and preserves the machine configuration and the state relationship.

```
lemma comp_WhileTrue_correct:
  "[|bval b s = Some True;
    (p1, s) => Some s'';
    !! ta. [|c1 = comp_com p1;
          code_installed ta c1;
          machine_config ta;
          Some s1 ~= None;
          state_rel s ta|] ==>
            EX ka ta'. steps ta ka = ta' &
              machine_config ta' &
              state_rel (the (Some s1)) ta' &
              REG ta' = (REG ta)(RName_0usr := REG ta' RName_0usr,
                                 RName_1usr := REG ta' RName_1usr,
                                 RName_2usr := REG ta' RName_2usr,
                                 RName_PC := REG ta RName_PC + 4 *
↪   word_of_int (int (length c1)));
    (While b p1, s1) => Some y;
    !! tb. [|code_installed tb c;
          machine_config tb;
          Some y ~= None;
          state_rel s1 tb|] ==>
            EX kb tb'. steps tb kb = tb' &
              machine_config tb' &
              state_rel (the (Some y)) tb' &
              REG tb' = (REG tb)(RName_0usr := REG tb' RName_0usr,
                                 RName_1usr := REG tb' RName_1usr,
                                 RName_2usr := REG tb' RName_2usr,
                                 RName_PC := REG tb RName_PC + 4 *
↪   word_of_int (int (length c)));
    code_installed t c;
    machine_config t;
    state_rel s t;
    c = comp_com (While b p1)|] ==>
      EX k t'. steps t k = t' &
        machine_config t' &
        state_rel (the (Some y)) t' &
        REG t' = (REG t)(RName_0usr := REG t' RName_0usr,
                         RName_1usr := REG t' RName_1usr,
                         RName_2usr := REG t' RName_2usr,
                         RName_PC := REG t RName_PC + 4 *
↪   word_of_int (int (length c)))"
  sorry
```

Listing A.4: The `comp_WhileTrue_correct` assumption. Assuming that `p1` executes correctly, we show that executing the instructions returned by `comp_com (While b p1)` in some state `s` produces some new state `y`, and preserves the machine configuration and the state relationship.