Certification of an Instruction Set Simulator

Xiaomu Shi

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Xiaomu SHI

Thèse dirigée par **Jean-François MONIN**
et codirigée par **Vania JOLOBOFF**

préparée au sein de **VERIMAG**
et de **Mathématiques, Sciences et Technologies de l'Information, Informatique**

Certification of an Instruction Set Simulator

Thèse soutenue publiquement le **10 juillet 2013**,
devant le jury composé de :

M. Yves Bertot
Directeur de Recherche, INRIA Sophia-Antipolis, Examinateur
Mme Sandrine Blazy
Professeur, IRISA, Rapporteur
M. Vania Joloboff
Directeur de Recherche, LIAMA, Co-Directeur de thèse
M. Xavier Leroy
Directeur de Recherche, INRIA Rocquencourt, Examinateur
M. Laurent Maillet-Contoz
Ingénieur, STMicroelectronics, Examinateur
M. Claude Marché
Directeur de Recherche, INRIA Saclay - Île-de-France et LRI, Rapporteur
M. Jean-François Monin
Professeur, Université de Grenoble 1 UJF, Directeur de thèse
M. Frédéric Rousseau
Professeur, Université de Grenoble 1 UJF, Examinateur

Table des matières

Table des figures

Liste des tableaux

Acknowledgements

First and foremost, I would like to express my sincere gratitude toward Prof. Jean-François Monin, my supervisor, for his guidance and kind support. He inspired me greatly during writing my thesis. I sincerely thank Mr. Vania Joloboff and the whole SimSoC-Cert team for rendering their help during the period of the project work. I also wish to give many thanks to my thesis reviewers Prof. Sandrine Blazy and Mr. Claude Marché for their patient reading and constructive suggestions, and to all the other jury members Mr. Yves Bertot, Mr. Xavier Leroy, Mr. Laurent Maillet-Contoz, and Prof. Frédéric Rousseau, who kindly accepted to take part in the jury. It has been a wonderful time and valuable opportunity to work in the FORMES group with people from all over the world sharing their knowledge and ideas. Special thanks to the Dr. Jianqi LI and the others from Tsinghua University for giving me helps in daily life and a good working environment. Last but not least I am grateful to my parents for their supports during my studies.

Abstract

This thesis introduces the work of certifying a part of a C/C++ program called `SimSoC` (Simulation of System on Chip), which simulates the behavior of architectures based on a processor such as ARM, PowerPC, MIPS or SH4.

A system on chip simulator can be used for software development of a specific embedded system, to shorten the development and test phases, especially when, as is the case for `SimSoC`, it offers a realistic simulation speed (about 100 Millions of instructions per second per individual core). Simulation makes it possible to reduce development time and development cost, allowing for co-design engineering, and possibility for the software engineers to run fast iterative cycles without requiring a hardware development board.

`SimSoC` is a complex software, including about 60,000 lines of C++ code, many complex features from SystemC library, and optimizations to achieve high simulation speed. The subset of `SimSoC` dedicated to the ARM processor, one of the most popular processor design, somehow translates in C++ the contents of the ARM reference manual, which is 1138 pages long. Mistakes are then unavoidable for such a complex application. Indeed, some bugs were observed even after the previous version of `SimSoC`, for ARMv5, was able to boot linux.

Then a critical issue is : does the simulator actually simulate the real hardware ? In our work, we aim at proving a significant part of the correctness of `SimSoC` in order to support the claim that the implementation of the simulator and the real system will exhibit the same behavior. Then a `SimSoC` user can trust the simulator, especially in very critical uses.

We focused our efforts on a critical part of `SimSoC` : the instruction set simulator of the ARMv6 architecture, which is considered in the current version of `SimSoC`.

Abstract

Approaches based on axiomatic semantics (typically, Hoare logic) are the most popular for proving the correctness of imperative programs. However, we prefered to try a less usual but more direct approach, based on operational semantics : this was made possible in theory since the development of an operational semantics for the C language formalized in Coq in the `CompCert` project, and allowed us to use the comfortable logic of Coq, of much help for managing the complexity of the specification. Up to our knowledge, this is the first development of formal correctness proofs based on operational semantics, at least at this scale.

We provide a formalized representation of the ARM instruction set and addressing modes in Coq, using an automatic code generator from the instruction pseudo-code in the ARM reference manual. We also generate a Coq representation of a corresponding simulator in C, called `Simlight`, using the abstract syntax defined in `CompCert`.

From these two Coq representations, we can then state and prove the correctness of `Simlight`, using the operational semantics of C provided by `CompCert`. Currently, proofs are available for at least one instruction in each category of the ARM instruction set.

During this work, we improved the technology available in Coq for performing *inversions*, a kind of proof steps which heavily occurs in our setting.

Résumé

Cette thèse expose nos travaux de certification d'une partie d'un programme C/C++ nommé `SimSoC` (Simulation of System on Chip), qui simule le comportement d'architectures basées sur des processeurs tels que ARM, PowerPC, MIPS ou SH4.

Un simulateur de *System on Chip* peut être utilisé pour developper le logiciel d'un système embarqué spécifique, afin de raccourcir les phases des développement et de test, en particulier quand la vitesse de simulation est réaliste (environ 100 millions d'instructions par seconde par cœur dans le cas de `SimSoC`). Les réductions de temps et de coût de développement obtenues se traduisent par des cycles de conception interactifs et rapides, en évitant la lourdeur d'un système de développement matériel.

`SimSoC` est un logiciel complexe, comprenant environ 60 000 de C++, intégrant des parties écrites en SystemC et des optimisations non triviales pour atteindre une grande vitesse de simulation. La partie de `SimSoC` dédiée au processeur ARM, l'un des plus répandus dans le domaine des SoC, transcrit les informations contenues dans un manuel épais de plus de 1000 pages. Les erreurs sont inévitables à ce niveau de complexité, et certaines sont passées au travers des tests intensifs effectués sur la version précédente de `SimSoC` pour l'ARMv5, qui réussissait tout de même à simuler l'amorçage complet de linux.

Un problème critique se pose alors : le simulateur simule-t-il effectivement le matériel réel ? Pour apporter des éléments de réponse positifs à cette question, notre travail vise à prouver la correction d'une partie significative de `SimSoC`, de sorte à augmenter la confiance de l'utilisateur en ce similateur notamment pour des systèmes critiques.

Nous avons concentré nos efforts sur un composant particulièrement sensible de `SimSoC` : le simulateur du jeu d'instructions de l'ARMv6, faisant partie de la version actuelle de `SimSoC`.

Résumé

Les approches basées sur une sémantique axiomatique (logique de Hoare par exemple) sont les plus répandues en preuve de programmes impératifs. Cependant, nous avons préféré essayer une approche moins classique mais plus directe, basée sur la sémantique opérationnelle de C : cela était rendu possible en théorie depuis la formalisation en Coq d'une telle sémantique au sein du projet `CompCert` et mettait à notre disposition toute la puissance de Coq pour gérer la complexitité de la spécification. À notre connaissance, au delà de la certification d'un simulateur, il s'agit de la première expérience de preuve de correction de programmes C à cette échelle basée sur la sémantique opérationnelle.

Nous définissons une représentation du jeu d'instruction ARM et de ses modes d'adressage formalisée en Coq, grâce à un générateur automatique prenant en entrée le pseudo-code des instructions issu du manuel de référence ARM. Nous générons également l'arbre syntaxique abstrait `CompCert` du code C simulant les mêmes instructions au sein de `Simlight`, une version allégée de `SimSoC`.

À partir de ces deux représentations Coq, nous pouvons énoncer et démontrer la correction de `Simlight`, en nous appuyant sur la sémantique opérationnelle définie dans `CompCert`. Cette méthodologie a été appliquée à au moins une instruction de chaque catégorie du jeu d'instruction de l'ARM.

Au passage, nous avons amélioré la technologie disponible en Coq pour effectuer des *inversions*, une forme de raisonnement utilisée intensivement dans ce type de situation.

Chapter 1

Introduction

1.1 Certification of SimSoC

This thesis describes the work that consists in certifying a part of a C/C++ program called `SimSoC` (Simulation of System on Chip) (26), which simulates the behavior of embedded systems architectures based on processors such as ARM, PowerPC, MIPS or SH4.

A system on chip simulator can be used for software development of a specific embedded system, to shorten the development and test phases, especially when, as is the case for `SimSoC`, it offers a realistic simulation speed (about 100 Millions of instructions per second per individual core). Simulation makes it possible to reduce development time and development cost, allowing for co-design engineering, and possibility for the software engineers to run fast iterative cycles without requiring a hardware development board.

Then a critical issue is: does the simulator actually simulate the real hardware? In our work, we aim at proving a significant part of the correctness of `SimSoC` in order to support the claim that the implementation of the simulator and the real system will exhibit the same behavior. Then a `SimSoC` user can trust the simulator, especially in very critical uses.

Considering only one module in `SimSoC`, namely the ARM simulator, it somehow encodes the 1138 pages of the ARM reference manual in C++. The whole simulator, which simulate ARM and PowerPC architecture, includes about 60,000 lines of C++ code. The software is very large and complicated with many complex features from

SystemC library, and optimizations to achieve high simulation speed. The first implementation of `SimSoC` ARM simulator was manually coded. Then, mistakes in the hand written code are unavoidable and difficult to find due to the complexity. Not only speed, but also *accuracy* is highly required. All simulated instructions must behave exactly like what is described in the specification (assuming the real hardware is conformant to the specification). From the experiments performed on `SimSoC`, bugs bringing a wrong behavior were observed from time to time but it was hard to reveal where they were. Using intensive tests can cover most of the instructions, but still left some untested rare cases of instructions, which lead to potential problems.

Therefore, a better approach is required to gain confidence in the correctness of the simulator. Our proposal is to certify the simulator `SimSoC` using formal methods.

In this thesis, we consider one of the modules in `SimSoC`: the ARM architecture simulator. ARM architecture is one of the most popular processor design in the embedded systems market, in particular mobile phones and tablets. As reported by ARM Holding company, 6.1 billion ARM-based processors have been brought to the market in year 2010 and 95% is used in the smart phone market.

As mentioned, the simulator is a large amount of software. And the specification itself is rather complex due to the rich architecture of ARM that consists of many components (to be detailed in Chapter 3). Before taking all features of the ARM simulator into account, we decided to focus on the basic parts, which are the most important and sensitive: the CPU part of the ARM architecture (such as used by the ARM11 processor family). At the time we started our work, the ARM simulator module implemented in `SimSoC` was the ARM Version 5 (ARMv5) architecture. Instead of applying certification for this old architecture, we decided to anticipate the evolution of `SimSoC` and to work on the next version: ARM Version 6 (ARMv6). For reasons explained below, related to the availability of proof technologies for C (especially `CompCert`), it was more convenient to have this module written in the C language rather than in C++. This module called `Simlight` (6) can run standalone, or as a component integrated in `SimSoC`. It is a simplified executable version of ARMv6 simulator.

More than 60% of the development size of `SimSoC` is in the CPU part, see Figure 1.1, the remaining parts consists in memory management (virtual memory and paging) interrupt handling and communications with peripherals. In summary, the complexity of the target could be reduced, but it still represents more than 10,000 lines of C code

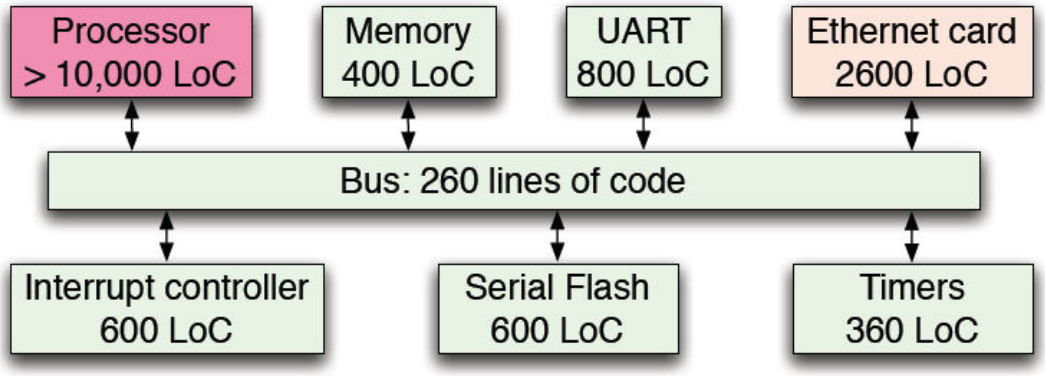

Figure 1.1: Development size of `SimSoC`

to be certified. Moreover, the complexity of the specification is invariant, as it is given by a heavy document, the ARM Version 6 Architecture reference manual.

Hence the issue at stake is how to certify a program of this size in relation to a rather complex specification.

Let us first recall that program certification is always based on a formal model of the program under study. Such a formal model is itself derived from a formal semantics of the programming language. For imperative programming languages such as C, a popular approach is to consider tools based on axiomatic semantics (Hoare logic), like `Frama-C` (13), a framework for a set of interoperable program analyzers for C. Most of the modules integrated inside rely on ACSL (ANSI/ISO C Specification Language), a specification language based on an axiomatic semantics for C. ACSL is powerful enough to express axiomatizations directly at the level of the C program. State labels can be defined to denote a program control point, and can then be used in logic functions and predicates. `Frama-C` is already quite a mature platform for C program static analysis and automatic deductive verification. An advantage of `Frama-C` or similar tools is that it is supported by automatic proof technologies, which save manpower consumption and make this approach quite convenient for the user. It was successfully applied to complex and tricky programs, e.g., Schorr-Waite algorithm, which deals with linked data structures.

`Frama-C` is able to perform:

– Analyzing the value of variables: `Frama-C` is able to compute and predict the range of numerical variables.

– Passing the proof obligations, (called *Verification Conditions*, VC for short) generated by `Why` (7) to automatic or interactive theorem provers.
– Slicing C program into shorter ones which preserve some properties.
– Navigating in C program.

However, in general, high automation tends to weaken the claims for certification, as automatic provers are themselves complex, then error-prone programs. In theory, such programs could produce certificates which could be checked by a reliable (e.g., LCF-based) proof assistant. But currently it is still far from being the case. An additional issue lies in the gap between the axiomatic semantics used and the real implementation, unless the VC generator is itself certified. This issue was considered recently, see below related work by Paolo Herms on `WhyCert` 1.4 – which was not available at the time we started to work on the certification of `SimSoC`.

Another important issue is that automation is possible only on theories or logics with limited expressive power. It can make it difficult to express specifications and expected properties at the right abstraction level, especially in a framework where the specification is very complex. Currently, `Frama-C` implements a superset of first order logic. An important current limitation for us is that ACSL is not able to describe pointer casting. On the contrary, the operational semantics defined for `CompCert` C (to be introduced below) is able to deal with any type casting.

The `Why` software is one of the most important components of `Frama-C` It is an implementation of Dijkstra's calculus of weakest preconditions. `Why` is the basis of the `Jessie` front-end, a plug-in of `Frama-C` which compiles ACSL annotated C code into the `Jessie` intermediate language. The result is given as input to the VC (Verification Conditions) generator of `Why`. `Why` then produces formulas to be sent to both automatic provers or interactive provers like Coq.

`Why` version 3 is a new and completely redesigned version of `Why`. It does not yet have its own front-end for C. It has become a standard library of logical theories (integer and real arithmetic, Boolean operations, sets and maps, etc.) and basic programming data structures (arrays, queues, hash tables, etc.). In order to transmit ACSL annotated C code to the `Why` 3 VC generator, `Jessie` generates an intermediate code in `WhyML`, which is a rich language for specification and imperative programming. In the new architecture, the specification language is enriched in order to support additional automatic provers. Furthermore, a formal interface is provided to facilitate the addition

of new external provers. Therefore, choosing `Why` or `Why 3` in our case would make us depend on the transformation chain provided by `Jessie` and `Why` together, from ACSL annotated C code to verification conditions for Coq.

In the case of `SimSoC`, we need to deal both with a very large specification including tricky features of the C language, such as type casting, which are used in tough functions related to memory management. In other words, we need a framework that is rich enough to make the specification manageable, using abstraction mechanisms for instance, in which an accurate definition of enough features of C is available. For the reasons explained above, it was unclear that `Frama-C` would satisfy those requirements, even with Coq as a back-end. Automated computations of weakest preconditions and range of variation are not relevant in our case. We need to verify more specific properties referring to a formal version of ARMv6 architecture. This specification is quite complex, for instance regarding the major data type to express the processor state (to be defined in Section 4.1).

In order to get the required flexibility and accuracy, we wanted to experiment a more direct approach based on a general proof assistant such as Coq. Fortunately, an operational semantics formalized in Coq of a large enough subset of the C language happened to be available from the `CompCert` project. We then decided to base our correctness proofs on this technology. Up to our knowledge, this is the first development of formal correctness proofs based on operational semantics, at least at this scale.

1.2 SimSoC

In this section, we introduce our certification target, `SimSoC`, a Simulator of System-on-Chip that can simulate various processors at a realistic speed. As a simulator of System-on-Chip, its objects are embedded system processors used in modern devices such as consumer electronics or industrial systems (e.g. ARM, PowerPC, MIPS). It is a so called *full system simulator* because it can simulate the entire hardware platform and run the embedded software "as is", including the operating system. Such kind of simulator plays an important role in embedded systems development, because the embedded software can be tested and developed on the simulator. In order to have software and hardware ready for the market at the same time, the software must by developed sometimes before the hardware is available. Then a executable model of the

SoC is required. A simulator also provides additional advantages combining simulation with usage of formal methods such as model checking or trace analysis to discover hardware or software bugs.

Our simulator, `SimSoC`, works at the low-level of the system. It takes real binary code as input and requires simulation models of the complete board: processor, memory units, bus, network controller, etc. It can emulate the behavior of instruction executions, exceptions, and peripheral interrupts. Other than software development, it may be used also for hardware design. When there are additional components provided by a third party, the software developers can test them in the full simulation environment with modularity.

`SimSoC` is developed in SystemC, which is itself a C++ library, and uses transaction level modeling (TLM) to model communications between the simulation modules. In order to simulate processors with a reasonably high speed, the instruction set simulation uses a technique named *dynamic translation*, which translates the binary input into an intermediate representation that is itself compiled into host code. Since `SimSoC` is a rather large and complex framework that influences the development of both hardware and software, we have to understand the most significant parts in order to be able to decide the certification object.

1.2.1 Instruction Set Simulation

A full system simulator must include the instruction set simulator, which reads the instructions of the program and exactly emulates the behavior of the target processor. In order to illustrate our certification target, we detail here the techniques to implement an instruction set simulator. There are three kinds of techniques implemented for `SimSoC` instruction simulation, that make trade-offs between accuracy and efficiency. They are: *interpretive simulation*, *dynamic translation with no specialization*, and *dynamic translation with specialization*. The *interpretive simulation* is the classical method, it includes the three stages: fetching, decoding and executing instructions. Although it is slow because of multiple redundant decoding phases, it is simple to implement and reliable. It is also used as the reference of performance for the other two techniques. The second and the third methods are based on dynamic translation, which uses an intermediate representation as the decoding result. Such intermediate representations of decoded instructions are stored into a cache and are re-used when

the same instructions are to be re-executed. The last method *dynamic translation with specialization* combines dynamic translation with *partial evaluation*. *Partial evaluation* specialization is a well known technique to optimize compilation of programs. The idea is to translate a program P which applies on data d into a faster specific program Pd. One can use partial evaluation in simulation to specialize one instruction into a simpler instruction, based on data known at decoding time. The `SimSoC` decoder implements partial evaluation. At decoding time, the dynamic translation maps the binary instructions to their partial evaluation specializations. Although specialization of instructions results into more memory usage, it is reasonable small compared to the memory size available on host machines nowadays.

The technologies used in `SimSoC` instruction set simulation are detailed in (27).

1.2.2 Performances

The ARM module of `SimSoC` used to implement the ISS of ARMv5 architecture was manually coded. The simulator is able to simulate the commercial System-on-Chip SPEAr Plus600 circuit from ST Microelectronics which is a dual core system based on over forty additional components, as well as the Texas Instruments AM1705 circuit. The simulator is able to emulate the interrupt controller, the memory controller, the serial memory controller, the Ethernet controller, and all peripherals which are necessary to boot Linux. Therefore, running the Linux kernel on the SPEAr Plus simulator module is a way of testing and debugging the simulator. First it reads the compressed Linux kernel binary from serial memory, uncompresses it, then starts Linux. The booting process takes only several seconds. The Ethernet controller can connect several simulators of the same SoC running on the same machine or not, through TCP/IP protocol. In `SimSoC`, a mature simulator for the ARMv5 architecture has been completed before starting our project, and two instruction set simulators for PowerPC and MIPS were also developed.

1.2.3 From ARMv5 to ARMv6

For this thesis, we decided to consider the next version (ARMv6) of the ARM architecture, which represented a step up in performance from ARMv5 cores. ARMv6 is essentially backward compatible with ARMv5. Here are the new features of ARMv6 architecture.

– The instruction set has been enlarged with extra instructions in six areas: media instruction, multiply instruction, control and DSP instruction, load/store instruction, architecture undefined instruction, and unconditional instruction. Fortunately, all ARMv5 mandatory instructions are ARMv6 mandatory instructions too. For simulator users, application code compiled with compilers for ARMv5 can be run over the ARMv6 simulator. If application users want to benefit from the new V6 instructions, they need to re-compile the code in the new environment.
– The Thumb mode has changed. Thumb instructions of ARMv5 are not portable to Thumb2 (ARMv6+), nor completely backwards compatible.

1.3 Outline

In summary, our goal is to certify (a part of) `SimSoC`, a system on chip simulator, using a framework based on the operational semantics of C formalized in Coq, and Coq itself. This thesis explains our achievements in this respect. Subsection 1.4 discusses relevant projects which use formal methods in the area of hardware processors, and why we need a new approach in our experiment. The contributions of our work are outlined in Subsection 1.5. Next, Chapter 2 provides the background on the main technologies used in our project, including a brief introduction to operational semantics, to Coq, and to the `CompCert` project. Our certification basis is the formal model of the ARMv6 instruction set simulator, which is described in Chapter 3. Then the certification target, a C program for simulating the ARMv6, is introduced in Chapter 4. The next chapter explains how repetitive and potentially error-prone tasks in the production of the two previous models are automated (Chapter 5). Chapter 6 describes how correctness proofs are carried out. In order to improve the performance and the management of the proofs, we had to develop a key proof technique for the "inversion" of assumptions related to the operational semantics of C expressions, which is described in Chapter 7. Chapter 8 is dedicated to an additional work using exhaustive testing for checking the coverage and correctness of the simulator decoder. Then we conclude in Chapter 9 and outline future research prospects.

1.4 Related work

The main difference between `SimSoC-Cert` (the certification of `SimSoC`) and the following projects is that we aim at proving the correctness of a hardware simulation whereas the target of the others is a certified hardware. The common point is that we need a formal specification of the instruction set of a specific processor architecture. Different proof assistants have been used to perform the certification on the formal model itself: Coq in our case; ACL2, HOL, etc. in other experiments. In our project, the formalization of the real chip ARMv6 is used as a reference for the behavior of an ARMv6 simulator written in C.

Using ACL2 for embedded systems

Researchers from Computational Logic, Inc., used ACL2 (A Computational Logic for Applicative Common Lisp) or Nqthm (Boyer-Moore Theorem Prover) (8) to specify and prove properties of several commercial processors as summarized in (10). ACL2 is a software system consisting of a programming language, an extensible theory in a first-order logic, and a mechanical theorem prover. It can act as both an automatic theorem prover and an interactive proof assistant. It supports automated reasoning in inductive logical theories, which is convenient for both software and hardware verification. Its programming language is a side-effect free extension of Common Lisp. And it is untyped. The base theory of ACL2 axiomatizes its programming language semantics and its built-in functions. User definitions in ACL2 programming language that satisfy a definitional principle extend the theory in a way that maintains the theory's logical consistency. The core of ACL2's theorem prover is based on term rewriting, and it is extensible in the following way: theorems discovered by the user can be used as ad-hoc proof techniques for subsequent conjectures.

Nqthm is a theorem prover sometimes referred to as the Boyer–Moore theorem prover. ACL2 system is essentially a re-implemented extension of Nqthm. We consider projects based on them together.

ACL2 and Nqthm are used to deal with different processor models in several projects. Among them, the work for Motorola's MC68020 is very close to ours (9). A large part of the user programming model of the MC68020 microprocessor is formalized as an abstract machine and its instruction operation as state transitions according

to its user's manual, which is similar to what we did for ARMv6. But the specification in Nqthm is formalized by hand; no automatic generator is used. The formal specification of the instruction set is defined in a functional way, as in our case. The target is the object code itself, considering that industrial strength compilers are not completely reliable; indeed, a certified compiler generating a machine code which is strictly equivalent to a high-level code was not available at that time. To keep the formalized system deterministic, they abandoned the instructions which may cause an *undefined* effect on the machine state. Comparing to this, we formalized our ARMv6 processor state differently in order to consider all the ARM instructions, including those instructions which produce an *undefined* state (see Section 3.2.1). The specification is written in the logic of Nqthm and proofs are obtained using the Nqthm fully automatic theorem prover, which cannot interact with the user once it is started. This is totally different from the Coq proof assistant we used, where proofs are designed in constant interaction with the user. The main theorems state that the execution of object code terminates and the returned machine state is considered normal, that all registers are set to the right location and that the effect of the execution is only on the relevant memory blocks.

The use of fully automatic theorem provers like Nqthm requires less man-power on theorem proving but much work on the specification. Once the formalization and the stated theorems are put in Nqthm logic system, we expect the automatic prover to do the rest. With this technique the object code produced by GNU C compiler for hundreds lines of C could be mechanically verified.

Still, the gap between C and the GNU C produced object code is not solved, because the GNU C compiler can not be considered fully trusted. So this method cannot be used for providing results which would be meaningful (and conveniently expressed) at the level of a C program.

Another project used ACL2 on the object processor CAP from Motorola company. The full ALU and I/O system of CAP is formalized in ACL2. The CAP implementation is a three-stage instruction pipeline; an appropriate correspondence between the CAP model with pipe-line and a simpler pipeline-free one is proven. Also the implemented DSP algorithm FIR (Finite Impulse Response) is proved to be equivalent to the formal specification. Moreover, the basic library for ACL2 on data structure was enriched, for example with array, list, record, bit vector, etc. New modules on modular arithmetic,

integers, hardware operations, and so on were built. These developments are reusable by other projects because the definitions are not specific to the CAP model.

A very recent work used ACL2 for developing a significant subset of X86 instruction set (23). The formal model of the X86 processor is executable and can run some binary programs. Some small binary programs are automatically verified under ACL2's interactive theorem proving environment using a symbolic execution technique.

Formalized x86 in Coq

In a recent work done in a cooperation between Harvard University, Lehigh University, NSF, and Google, a model of x86 has been formalized in Coq (47). The whole project is called Native Client (NaCl). It is a platform allowing Google's Chrome browser users to execute native code on the browser. A sandbox policy is used to ensure several properties. The most important ones are to ensure that read/write operation on arbitrary memory blocks are only caused by trusted code, instructions related to system calls are avoided, and communication can only happen within well-defined entry points. These properties of the sandbox policy protect the system from bugs or concurrent access to memory. The aim of this work is to obtain a highly trusted assurance checker for the sandbox policy. A checker for a 32-bit x86 (IA32) processor without floating-point instructions, RockSalt was built. This new achievement is better than the original version provided by Google in three aspects: it is faster, lighter, and more flexible.

This project has some similarities with ours: the core of RockSalt was automatically generated from the Coq formal specification using extraction to OCaml code; then it was manually translated in C language in order to have an implementation for NaCl. A new Coq model for x86 was defined. In the future, it is expected to support reasoning about the behavior of x86 machine code using a verified compiler such as `CompCert`. RockSalt relies only on a DFA (Deterministic Finite-state Automaton) encoding table and some trusted C code. The checker is extremely fast: 1M instructions can be checked within one second. Also, it has a small run-time trusted computing base and can be integrated into NaCl runtime easily. The formal model of x86 ISS (Instruction Set Simulator) has the same architecture as a translator: a decoder from binary code to an abstract syntax, a compiler from this abstract syntax to RTL instructions; and an interpreter for RTL instructions. Because no predefined formal model could be

considered as a trust-base, tests were created for this complex model, in order to gain confidence.

For validation, an executable OCaml model is extracted from the Coq specification, and the behavior of the OCaml model is compared with the real x86. More than ten million binary instructions could be simulated in around sixty hours on Intel Xeon. Moreover, fuzz tests are used to cover problems that cannot be considered by the previous tests. Fuzz test can cover all forms of one kind of instruction, and all of them can be exercised. The Coq processor model in this project is essentially an intermediate specification which is used to obtain more secure C code. However the manual translation from extracted OCaml code to C is not explained in (47). Tests are only used at the level of OCaml code. Altogether, it is unclear that the resulting C code has the same behavior as the Coq specification.

An instruction set generator for an ideal processor

A German project describes in (32) an Instruction Set Simulator generation technique which aims at generating an efficient ISS from the RTL (Register Transfer Level) description. The ITL (Iterative Temporal Logic) language is used to design the ISS at the RTL level, then C++ code is generated from it. The interval temporal logic is a combination of temporal logic and first-order logic able to deal with sequential and parallel composition. It includes a notion of finite sequences. First order interval logic was first designed for formalizing and verifying hardware protocols. It is sufficient for specifying computer-based systems, both hardware and software. Some of the standard operations of VHDL or Verilog language can be expressed as temporal logic expressions to describe the behavior of a synchronous sequential system.

The verification of safety properties is performed using a technique called IPC (Interval Property Checking) which is designed to check if a model satisfies a set of properties written in a dedicated verification language, ITL in this case. The main idea followed here is to use an arbitrary initial state instead of the initial state of BMC (Bounded Model Checking). Any property that holds from an arbitrary initial state also holds from any reachable state. Conversely, false negatives can occur in IPC. These false negatives need to be removed by adding invariants in order to restrict the set of initial states. In order to gain speed for simulation, optimizations are performed on the C++

code generation like in any other state-of-the-art simulators by the following two methods: first, arithmetic and simple logic operation are mapped to corresponding native C++ operations; second, additional variables are used to store temporary results and to cache intermediate results of computations. The formal verification aims at proving the equivalence between RTL and ISA models. The main idea of the equivalence proof is very similar to ours (see Chapter 6): it is based on a mapping from the RTL state representing CPU to the corresponding ISA state in C, and on next-state transitions for both the RTL model and the ISA model of the C program. Proving that the interface signals of the design is correctly implemented is also performed in the same way. For code generation, the input source is specified in VHDL. The first experiment was achieved on an invented ideal simplified CPU model. It contains eight 16-bit registers and a special register used as interrupt return vector. There are only seven instructions for logic, arithmetic, memory accessing, and jump. For the second experiment, an industrial processor design was chosen. It contains 64 32-bits registers and 88 instructions based on the DLX instruction set architecture and a memory model with simple interface. This corresponds to 10,000 lines of VHDL code, reformulated into a 2000 lines ITL specification to be used as the source of the C++ code generator. Compared to `SimSoC`, the size of the formal specification is smaller and with ITL specifications, less properties can be expressed and proved than with Coq.

A trustworthy monadic formalization of ARMv7 instruction set architecture

The computer laboratory in Cambridge University used HOL4 to formalize the instruction set architecture of ARMv7 (21). The target here was to verify an implementation of this architecture with *logical gates*, whereas for `SimSoC`, we consider a simulation written in C. Reusing the work done at Cambridge was considered for `SimSoC`. However, as our approach is based on `CompCert`, which is itself written in Coq instead of HOL4, it was more convenient to write our own specification. The achievements obtained at the Cambridge projects are:

– A model of the ARM instruction set architecture in the HOL language. Other than ARMv7 instruction set which is backward compatible with previous versions, the model considered here has the full support for the Thumb-2 instruction set, too.

- Additional software for simulation, an assembler, and a disassembler in ML.
- Automatic extraction of single step theorems from a monadic representation of a single ARM instruction, for evaluation.
- Besides the tools and specifications formalized or derived inside HOL logic, some other tools that are outside HOL logic: the assembly code parser, the disassembler, and the encoder. They are considered not completely reliable.
- The formalized model can operate on machine code level.
- The execution results are used for comparison with the real hardware in order to validate the model.

The instructions have been grouped together with related ones. This reduces greatly the specification size. But the HOL 4 model is a little too precise. For example, it specifies the resource accessing order when updating the PC. But this order is not specified in the reference manual. During validation, three boards are used to assess the execution results. And the tests are generated randomly by a test generator. Additional confidence in this development was achieved by observing the behavior of verified code, and running the model on ARM code that calculates a non-trivial known function. Some weaknesses are: storing instructions are not completed; problems happen when updating on registers PC or SP occur; exceptions are not well handled; tests do not cover unpredictable and undefined instruction, which have been filtered in test generation; the model does not include the privileged mode, nor the instruction changing the current processor mode; implementation dependence or system features cannot be fully tested.

WhyCert: A certified implementation of VC generator

Paolo Herms's Ph.D thesis (28) (29) provides a certified verification condition generator for several provers, called `WhyCert`. The work is also based on `CompCert` C operational semantics. Using a VC generator is another way of ensuring the safety requirements of programs written in high-level programming languages, as mentioned in Section 1.1, as well as the importance of certifying a VC generator. In `WhyCert`, the VC generator was implemented and proved sound in Coq, then extracted to an executable OCaml program. As suggested by the name `WhyCert`, this work is inspired by `Why` (20), a platform for deductive program verification. To make it usable with arbitrary theorem provers as back-ends, it is generic with respect to a logical context, containing

arbitrary abstract data types and axiomatisations. This is suitable to formalized memory models needed to design front-ends for mainstream programming language, as it is done for C by VCC above Boogie or by `Frama-C/Jessie` above `Why`. The inputs are imperative programs written in a core language which operates on mutable variables which are typed in the logical context. The output is made of logic formulas which are built upon the same logical context.

The seL4 project

The NICTA company provides a secured microkernel seL4 (31), a member of L4 microkernel family. The formal verification is based on the interactive theorem prover Isabelle/HOL. The correctness proofs are established according to a correspondence between an abstract and a concrete representations of the seL4 system. The concrete model is the C implementation, which is translated to Isabelle using a intermediate language called `com`, which has an operational semantics like `CompCert` C. However this language handles a smaller subset of C than `CompCert` C. Unsupported features of `com` that are supported in `CompCert` C include:

- pointers to automatic storage
- float, function pointer, union
- switch, goto, break, continue

Note that for SimSoC, we need function pointer, switch, break.

1.5 Contributions

In this work we developed a correctness proof of a part of the hardware simulator `SimSoC`. This is not only an attempt to certify a simulator, but also a new experiment on the certification of non-trivial programs written in C. In our approach, we do not use the popular axiomatic semantics, but the C operational semantics defined by the `CompCert` project.

We provide a formalized representation of the ARM instruction set and addressing modes in Coq, using an automatic code generator from the instruction pseudo-code in the ARM reference manual. We also generate a Coq representation of a corresponding simulator in C, called `Simlight`, using the abstract syntax defined in `CompCert`. The

text version of `Simlight` was previously developed as a component of `SimSoC` by C. Helmstetter (6).

From these two Coq representations, we can then state and prove the correctness of `Simlight`, using the operational semantics of C provided by `CompCert`. Our first achievement in this direction was described in (57). Currently, proofs are available for at least one instruction in each category of the ARM instruction set.

During this work, we improved the technology available in Coq for performing *inversions*, a kind of proof steps which heavily occurs in our setting (46).

Additional contributions include a generator of a decoder for ARM instructions, also based on the analysis of the ARM reference manual, and a test generator for the instruction decoder, which can generate massive tests covering all ARM instructions.

Chapter 2

Background

This chapter provides a short introduction to the scientific background of our work: operational semantics, to define the meaning of programs; the Coq proof assistant, which is used to define the formal model of ARMv6 architecture and to perform correctness proofs; finally `CompCert`, which contains an operational semantics of C formalized in Coq, and is then the basis of the formal model that we use for the instruction set simulator `Simlight`. We pay a particular attention on underspecified behaviours: this happens when different compilation strategies may provide different behaviours for the same program, as is the case for C. Such issues are illustrated on a very simple toy language, *ese*.

Résumé

Ce chapitre contient une courte introduction au cadre scientifique dans lequel notre travail a été developpé. On commence par quelques notions de sémantique opérationnelle, permettant de définir la signification des programmes. On présente ensuite l'assistant à la preuve Coq, que nous avons utilisé pour définir notre modèle formel de l'architecture ARMv6 et d'effectuer des preuves de correction. Nous terminons par `CompCert`, qui fournit notamment une sémantique opérationnelle de C formalisée en Coq – c'est l'ingrédient essentiel que nous utilisons pour produire un modèle formel du simulateur d'instructions `Simlight`.

Une attention particulière est portée aux comportement sous-spécifiés : cela se produit lorsque différentes stratégies de compilation peuvent aboutir à des comportements

différents pour un même programme, ce qui est le cas avec le langage C. Pour illustrer ce genre de problèmes, nous introduisons un langage jouet, *ese*, contenant des expressions avec effet de bord.

2.1 Operational Semantics

In computer science, there are three traditional ways to express how programs perform computations: axiomatic semantics, denotational semantics, and operational semantics. Formal semantics are important because it can give an abstract, mathematical, and standard interpretation of a programming language. It helps to understand what a program written in this language does and to verify what we expect from the program. In a few words:

- Denotational semantics constructs mathematical objects which describe the meaning of expressions of the language using stateless partial *functions*. All observably distinct programs have distinct denotations.
- Operational semantics is more concrete because it is based on states. However, in contrast with a low-level implementation, operational semantics considers abstract states. The behavior of a piece of program corresponds to a transition between abstract states. This transition relation allows us to define the execution of programs by a mathematical computation *relation*. This approach is quite convenient for proving the correctness of compilers, using operational semantics for the source and target languages (and, possibly intermediate languages). Operational semantics is used in `CompCert` to define the execution of C programs, or more precisely programs in the subset of C considered by the `CompCert` project. The work presented in this thesis is based on this approach.
- Axiomatic semantics describes the effect of programs by assertions. A well-known example is Hoare logic. It is one of the most popular approaches for proving the correctness of programs.

A good tutorial on programming language semantics is Benjamin C. Pierce's *Software Foundation*[1]. It is mainly dedicated to operational semantics and it contains an introduction to Hoare Logic. The material presented in this tutorial is formalized in

1. http://www.cis.upenn.edu/ bcpierce/sf/

the Coq proof assistant. Another interesting introduction can be found in (48). It is more detailed than *Software Foundation*, but it is not supported by a proof assistant.

Operational semantics can be used to reliably prove results on a programming language. Operational semantics can be presented in two styles. Small-step semantics, often known as structural operational semantics, is used to describe how the single steps of computations evaluate. The other is big-step semantics, or natural semantics, which returns the final results of an execution in one big step. The corresponding transition relation is defined by rules, according to the syntactic constructs of the language, in a style which is inspired by natural deduction.

The book (48) explains that the choice between small-step semantics and big-step semantics depends on the objective. They sometimes can be equivalent. But in general, they provide different views of the same language and we have to choose an appropriate one for a particular usage. Moreover, some language constructs can be hard or even impossible to define with one of these semantics whereas it is easy with the other style. In general, when big-step semantics can be used, it is simpler to manage than small-step semantics.

In order to illustrate some issues on operational semantics and its different flavors which are important for us, let us consider a simple language called *ese*, for expressions with side-effects. This language allows us to present some typical issues of C language, related to the the evaluation order of expressions and statements. The ISO-C standard that mentions the evaluation order of expressions with side-effect on the same object is undefined, for example:

```
i = ++i + 1;
a[i++] = i;
```

Several orders are allowed for each of the previous assignments, because they include two side effects on variable `i` – according to ISO-C standard, there are two "sequence points" in them.

Other examples are given by Brian Campbell in the CerCo project (12), in order to show that the evaluation order constraints in C are very lax and not uniform.

```
x = i++ && i++;
x = i++ & i++;
```

Our toy language *ese* is designed to illustrate similar issues. The constructs of *ese* are: constants $C\ n$, where n is a natural number, a unique variable V, the addition $P\ ese\ ese$ of two arguments of type *ese*, and the assignment of the variable with a value expressed by an *ese*. Its abstract syntax is as follows.

$$ese\ ::=\ C\ n \mid V \mid P\ ese\ ese \mid A\ ese$$

Figure 2.1: Syntax of toy language *ese*

The semantics in big-step style is inductively defined using the following rules. The parameter *state* of type natural number is introduced here to store the current value of V. After an evaluation, a new *state* is returned. The evaluation takes an initial state and an expression to compute, and returns a new state and a natural number which is the evaluation result. The notation $\xrightarrow{bs}$ means "evaluates to".

$$\frac{}{st,\ C\ n\ \xrightarrow{bs}\ st',\ n} \tag{2.1}$$

$$\frac{}{st,\ V\ \xrightarrow{bs}\ st,\ st} \tag{2.2}$$

$$\frac{st,\ e_1\ \xrightarrow{bs}\ st',\ n_1 \qquad st',\ e_2\ \xrightarrow{bs}\ st'', n_2}{st,\ P\ e_1\ e_2\ \xrightarrow{bs}\ st'',\ (n_1\ +\ n_2)} \tag{2.3}$$

$$\frac{st,\ e\ \xrightarrow{bs}\ st',\ n_1 \qquad n_1,\ V\ \xrightarrow{bs}\ st'', n_2}{st,\ A\ e\ \xrightarrow{bs}\ st'',\ n_2} \tag{2.4}$$

Figure 2.2: Big-step operational semantics of the toy language *ese*

Rule 2.4 is for assignment. A simpler and equivalent version is:

$$\frac{st,\ e\ \xrightarrow{bs}\ st',\ n}{st,\ A\ e\ \xrightarrow{bs}\ n,\ n} \tag{2.5}$$

The version given in rule 2.4 is closer to the small-step semantics to be presented later, which exposes an explicit evaluation order. To this effect, the assignment is split into two parts: evaluating the right-hand side then putting the result into the left-hand side.

For instance, from the state where `V` contains 0, the expression in C syntax

```
V + ((V = 1) + (V = 2))
```

evaluates to 3, with a final state where `V` contains 2. This expression is formalized by the term $P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2)))$, and the previous statement is formalized by:

$$0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) \xrightarrow{bs} 2, 3.$$

This statement is proved by systematic applications of the rules given in Figure 2.2. The proof is driven by the shape of the expression. Each constructor (C, V, P, A) is handled by a specific rule and leads to premises with smaller expressions (in this language), which means that the execution will terminate for any expression. Moreover, the semantics defined here is deterministic; the evaluation order is leftmost and innermost. This is expressed by the following lemma:

Lemma 2.1. *If* $st,\ t \xrightarrow{bs} st'\ v$, *and* $st,\ t \xrightarrow{bs} st''\ v'$, *then* $v = v'$ *and* $st = st''$.

Using big-step semantics, we can also describe a non-deterministic system by adding one rule for right to left evaluation to offer another evaluation order:

$$\frac{st,\ e_2 \xrightarrow{bs} st',\ n_2 \qquad st',\ e_1 \xrightarrow{bs} st'', n_1}{st,\ P\ e_1\ e_2 \xrightarrow{bs} st'',\ (n_1\ +\ n_2)} \tag{2.6}$$

Then the output of the evaluation cannot be predicted: the same expression can return different states and results. For instance, we have

$$\begin{aligned}
0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) &\xrightarrow{bs} 2, 3\\
0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) &\xrightarrow{bs} 1, 3\\
0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) &\xrightarrow{bs} 2, 5\\
0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) &\xrightarrow{bs} 1, 4
\end{aligned}$$

Next, the following description gives the small-step operational semantics rules of the same toy language. This time, the small-step rules take an expression of type *ese* and the initial state which stores the current value of variable V, and return the reduced expression and the new state. The symbol $\xrightarrow{ss}$ means "reduces to in one small step".

In small-step semantics, two rules ((2.9) and (2.10)) are needed to define the leftmost and innermost evaluation order. And there is no rule for reducing a single constant. From the number of rules, we see that the definition of deterministic computations with a given evaluation order is more complex with small-step operational semantics than with big-step semantics.

$$\frac{}{V,\ st \xrightarrow{ss} (C\ st),\ st} \tag{2.7}$$

$$\frac{}{(P\ (C\ n_1)\ (C\ n_2)),\ st \xrightarrow{ss} (C\ (n_1\ +\ n_2)),\ st} \tag{2.8}$$

$$\frac{e_1,\ st \xrightarrow{ss} e_1',\ st'}{(P\ e_1\ e_2),\ st \xrightarrow{ss} (P\ e_1'\ e_2),\ st'} \tag{2.9}$$

$$\frac{e_2,\ st \xrightarrow{ss} e_2',\ st'}{(P\ (C\ n_1)\ e_2),\ st \xrightarrow{ss} (P\ (C\ n_1)\ e_2'),\ st'} \tag{2.10}$$

$$\frac{}{(A\ (C\ n)),\ st \xrightarrow{ss} V,\ n} \tag{2.11}$$

$$\frac{e,\ st \xrightarrow{ss} e',\ st'}{(A\ e),\ st \xrightarrow{ss} (A\ e'),\ st'} \tag{2.12}$$

Figure 2.3: Small-step operational semantics of the toy language *ese*

We can also have a non-deterministic small-step semantics by modifying one of the rules of the plus operation to remove the leftmost and innermost order: changing rule (2.10) in Figure 2.3 into:

$$\frac{e_2,\ st \xrightarrow{ss} e_2',\ st'}{(P\ e_1\ e_2),\ st \xrightarrow{ss} (P\ e_1'\ e_2),\ st'} \tag{2.13}$$

Considering the set of possible executions allowed by the non-deterministic semantics, we have more results by using small-step semantics than using big-step semantics. Taking the same example $P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2)))$ as above, the possible executions in small-step semantics are:

$$0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) \xrightarrow{ss} 3, 1$$
$$0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) \xrightarrow{ss} 3, 2$$
$$0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) \xrightarrow{ss} 4, 1$$
$$0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) \xrightarrow{ss} 5, 2$$
$$0, P\ V\ (P\ (A\ (C\ 1))\ (A\ (C\ 2))) \xrightarrow{ss} 6, 2$$

The last result is obtained by performing the assignment $A\ (C\ 1)$, then the assignment $A\ (C\ 2)$; at this point, the value stored in the state equals 2. Next, performing plus in any order will compute the result of 6 and the state still stores 2. On the other hand, the big-step semantics fails to express that 6 can be returned.

In contrast with big-step semantics, the sequence corresponding to an assignment (evaluation the right-hand side, then putting the result into the left-hand side) can actually be interrupted when we consider small-step semantics, and the evaluation of another sub-expression can then occur.

In general, big-step semantics is not the right approach for dealing with non-deterministic executions or under-specified semantics, because it is not able to cover all the possible execution cases.

Note that `CompCert` includes a big-step deterministic semantics and a small-step non-deterministic semantics for `CompCert` C.

2.2 Coq

2.2.1 Short introduction

Coq(5) is an interactive theorem prover, implemented in OCaml. It allows the expression of mathematical assertions, mechanically checks proofs of these assertions, helps to find formal proofs, and extracts a certified program from the constructive proof of its formal specification. Coq can also be presented as a dependently typed λ-calculus (or functional language). Here we just illustrate the syntax on simple examples. For a detailed presentation, the reader can consult (15) or (5).

- $fun\ (n : nat) \Rightarrow n$ is the identity function on natural numbers; its type is written as $nat \rightarrow nat$. Function application is not written as $f(x)$ but $f\ x$, or $(f\ x)$ if grouping is needed. With several arguments, the syntax is $f\ x\ y$ or $(f\ x\ y)$ instead of $f(x,\ y)$.
- We can write definitions as follows:

  ```
  Definition idn := fun (n : nat) ⇒ n.
  ```

  An equivalent and more common syntax for this definition is:

  ```
  Definition idn (n : nat) := n.
  ```

  For instance, the application of idn to 3 is written $(idn3)$ and this term reduces to 3.

– $fun\ (X\ :\ Type)\ (n\ :\ X)\ \Rightarrow\ n$ is the *polymorphic* identity function on an arbitrary type X; its type is written $\forall\ X\ :\ Type,\ X\ \rightarrow\ X$.

```
Definition id (X : Type) (n : X) := n.
```

Note that it expects 2 arguments, for instance, we can write ($id\ nat\ 3$). Like most of functional programming languages, Coq can also perform type inference. If we define id as following:

```
Definition id {X : Type} (n : X) := n.
```

The application can be just written as $id\ 3$. Coq can get the explicit X from the type of 3.

– A dependent type is a type that depends on a value. It is very flexible to use, as to refine the type of a function without including the whole specification. A very simple example is to define a predecessor with only the rule for case 0:

```
∀ n : nat, n > 0 → nat
```

– Coq also includes inductive types, as explained in the next subsection.

A proof term of type $\forall\ n\ :\ nat,\ P\,n\ \rightarrow Q\,n$ is $fun\ (n : nat),\ P\ n\ \rightarrow\ Q\ n$ is a function which takes a natural number n and a proof of $P\ n$ as arguments and returns $Q\ n$. In general, proofs are functions and checking the correctness of a proof boils down to type-checking.

Coq is not an automated theorem prover: the logic supported by Coq (CiC [1]) includes arithmetic; therefore it is too rich to be decidable. However, type-checking (in particular, checking the correctness of a proof) is decidable. As full automation is not possible for finding proofs, human interaction is essential. The latter is realized by *scripts*, which are sequences of commands for building a proof step by step. Coq also provides built-in tactics implementing various decision procedures for suitable fragments of CiC and a language called `Ltac` which can be used for automating the search of proofs and shortening scripts.

1. Calculus of Inductive Constructions.

2.2.2 Inductive definitions

To make a better illustration, we use the same toy language *ese* 2.1 as in the previous section. Here we show how to inductively define its syntax and its big-step operational semantics:

```
Inductive tm : Type :=
  | C : nat -> tm      (* constant *)
  | V : tm             (* the unique variable *)
  | P : tm -> tm -> tm (* plus *)
  | A : tm -> tm       (* assignment *)
```

An inductive definition can handle recursive specifications of types; it defines how it is constructed. The type `tm` is the type of the toy language *ese* which can be a constant (constructor `C` associated with a natural number of type `nat`), a (unique) variable (constructor `V`), or one of the following two operations: plus (two expressions of type `tm` connected by the constructor `P`) or assignment (constructor `A` with an expression of type `tm` as input)

Then the inductive definition below gives the annotated inductive type to describe the deterministic evaluation relation of the corresponding *ese* in big-step style. The type of the evaluation `eval` is a relation, describing the transition from an input expression `tm` and a state to a new state and an evaluation result of type `nat`, a natural number. Each clause is defined according to a rule in Figure 2.2.

```
Inductive eval : state -> tm -> state -> nat -> Prop :=
  | E_Const : forall s n,
      eval s (C n) s n
  | E_Var : forall s,
      eval s V s s
  | E_Plus : forall s t1 n1 s' t2 n2 s'',
      eval s t1 s' n1 ->
      eval s' t2 s'' n2 ->
      eval s (P t1 t2) s'' (n1 + n2)
  | E_Assign : forall s s' s'' t n1 n2,
      eval s t s' n1 ->
      eval n1 V s'' n2 ->
      eval s (A t) s'' n2.
```

2.2.3 Proofs and tactics

In order to show a concrete proof using Coq proof assistant, we recall Lemma 2.1 which claims the big-step operational semantics of *ese* is deterministic. we first formalize the corresponding statement as follows:

```
Lemma eval_deterministic:
  forall st t st' st'' v v',
  eval st t st' v ->
  eval st t st'' v' ->
  (v = v') ∧ (st' = st'').
```

It states that, with the same initial state `st` and expression `t`, evaluating the big-step semantics defined in Figure 2.2 will return the same results and the same new states. Then we use Coq in an interactive way to verify this statement. The general idea is to make an induction on `eval st t st' v`, name as hypothesis `ev1`. According to the rules in the inductive definition of `eval`, there are four cases to consider. Under each case of `ev1`, we also have to consider the corresponding derivation of hypothesis `ev2` of type `eval st t st" v'`. The proof script contains a sequence of tactics to lead Coq to perform all these steps, checking the correctness of the claims we made. Here is a short explanation on some basic and frequently used tactics:

- `intros` moves the quantifiers and hypotheses from the goal to the context of assumptions.
- `induction` does case analysis for inductively defined types. Induction hypotheses are automatically put into context.
- `inversion` derives the constraints on variables according to the inductive definition corresponding to the hypothesis that is inverted.
- `reflexivity` checks that the left-hand side and the right-hand side of an equational goal are convertible.
- `rewrite` performs replacement according to an equational hypothesis.

The following code from `Proof` to `Qed` provides a formal proof of the determinism of big-step semantics of *ese* stated above.

```
Proof.
  intros until v'; intros ev1 ev2.
  generalize dependent v'.
```

```
  generalize dependent st''.
  induction ev1.
    (*Case "C"*)
    intros;
    inversion ev2; subst; split; try reflexivity.
    (*Case "V"*)
    intros;
    inversion ev2; subst; split; try reflexivity.
    (*Case "P"*)
    intros;
    inversion ev2; subst; split;
    apply IHev1_1 in H2; destruct H2 as [Heqn1 Heqst1];
    rewrite Heqst1 in IHev1_2;
    apply IHev1_2 in H5; destruct H5 as [Heqn2 Heqst2];
    [rewrite Heqn1; rewrite Heqn2; reflexivity | exact Heqst2].
    (*Case "A"*)
    intros;
    inversion ev2; subst;
    apply IHev1 in H1;
    destruct H1; rewrite H; split; reflexivity.
Qed.
```

2.2.4 Interactive proof assistant vs automated theorem prover

An interactive proof assistant, such as Coq, requires man-machine collaboration to develop a formal proof. Human input is needed to create appropriate auxiliary defintions, choose the right inductive property and, more generally, to define the architecture of the proof. Automation is used for non-creative proof steps and checking the correcntess of the resulting formal proof. A rich logic can be handled in an interactive proof assistant for a variety of problems.

On the other hand, fully automated theorem provers were developed. They can perform the proof tasks automatically, that is, without additional human input. Automated theorem prover can be efficient in some cases. But problems appear to be inevitable: if we are able to automatically prove a formula, it means that it belongs to a decidable (or at least semi-decidable) class of problems. It is well-known that decidable logics are much less powerful, expressive and convenient than higher-order logic.

Then the range of problems we can model with an automated theorem prover is smaller than with an interactive proof assistant. In practice, both approaches are important in the fields of computer science and mathematical logic. Here in our project, a rich logical system is needed, in order to manage the complexity of the specification and of the proofs.

2.2.5 Applications

Georges Gonthier (of Microsoft Research, in Cambridge, England) and Benjamin Werner (of INRIA) used Coq to create a surveyable proof of the four color theorem, which was completed in September, 2004 (24) Based on this work, a significant extension to Coq was developed, which is called Ssreflect (which stands for "small scale reflection"). Despite the name, most of the new features added to Coq by Ssreflect are general purpose features, which is useful not merely for the computational reflection style of proof.

The same technology was then used for the formal verification of an important result from finite group theory, the "odd theorem". A simplified proof has been published in two books: (Bender & Glauberman 1995), which covers everything except the character theory, and (Peterfalvi 2000, part I) which covers the character theory. This revised proof is still very hard, and is longer than the original proof, but is written in a more leisurely style. A fully formal proof, checked with the Coq proof assistant, was announced in September, 2012 by Georges Gonthier and fellow researchers at Microsoft Research and INRIA.(25)

`CompCert` (35) is a formally verified optimizing compiler for a subset of the C programming language which currently targets PowerPC, ARM and 32-bit x86 architectures. The compiler is specified, programmed, and proved in Coq. It aims to be used for programming embedded systems requiring reliability. The performance of its generated code is often close to that of gcc (version 3) at optimization level O1, and is always better than that of gcc without optimizations.

2.3 CompCert

In a previous section (Sec 6.1), we mentioned that we use results of the `CompCert` project in order to link the formal representation of ARMv6 architecture with the C

representation of this architecture in Simlight. Now we introduce CompCert in more detail. CompCert is a verified compiler for the C programming language provided by INRIA (35). It has a long translation chain of eleven steps, from C source code into assembly code. Every internal representation has its own syntax and semantics defined in Coq. It is formally verified in the following sense: the produced assembly code is proved to behave exactly the same as the input C program, according to a formally defined operational semantics of these languages.

The 2 first languages considered in the CompCert translation chain, CompCert C and Clight, are actually subsets of the C language. Like C, CompCert C is non-deterministic: for some expressions cause side-effect and have more than one evaluation order. On the other hand, all expressions of Clight are pure. Assignments and function calls in Clight are treated not as statements but as expressions. The reason why we choose CompCert C rather than Clight to represent Simlight is that it is much more user-friendly and convenient. Indeed, as Clight expressions are pure and deterministic, a number of auxiliary variables have to be introduced in order to manage intermediate states.

Here we present a small example of a C program to illustrate the last point. The original C code is as:

```
void main(int x, int y)
{
  int a;
  int b;
  int v;
  a = f(f1(v, f2(x, y)), f3(a, 1), f4(b, 3));
}
```

All the function calls (fx) are side-effect free operations. Then using CompCert compiler, we are able to generate the CompCert C and Clight representations. The CompCert representation is exactly the same as the original C code in this case. But the Clight representation is quite different, with the introduction of additional temporary variables (which are different from local variables, they do not reside in memory).

```
void main(int x, int y)
{
  int a;
```

```
  int b;
  int v;
  register int $5;
  register int $4;
  register int $3;
  register int $2;
  register int $1;
  $1 = f2(x, y);
  $2 = f1(v, $1);
  $3 = f3(a, 1);
  $4 = f4(b, 3);
  $5 = f($2, $3, $4);
  a = $5;
}
```

The proof based on these two representations can be expected to have the same complexity, because the complexity of the proof work is caused by the C memory model. Using either of them will face the same memory model (this will be detailed in Chapter 6). The transition corresponding to the evaluation of a given high-level expression (as the one given above) will anyway be decomposed in smaller transitions, either if we use the more complicated semantics of `CompCert` C on the original shorter expression, or if we use the simpler semantics of `Clight` on the corresponding longer `Clight` expression. Therefore, we don't expect a real gain in using `Clight` rather than `CompCert` C at the proof stage, while we would lose readability and convenience in the C code.

In `SimSoC-Cert`, we use two parts of `CompCert` C. The first is the `CompCert` basic library. It defines data types for words, half-words, bytes etc., and bitwise operations and lemmas to describe their properties. In our Coq model, we also use these low level representations and operations to describe the ISS (Instruction Set Simulator) model. The second is the `CompCert` C language (its syntax and semantics), from which we get a formal model of `Simlight`. In our correctness proofs, wa can then analyze its behaviour step by step and compare it with our Coq model of ARM.

2.3.1 CompCert library

In `CompCert`, a reusable basic library on machine integers (type `int`) and bitwise operations is formally defined in Coq. The type `int` is based on type `Z` from the Coq

standard library, with a proof to guarantee that the range of the value is between 0 and the modulus. Parameterized by `wordsize` of type `nat` (natural number), the integer module can be instantiated to `byte`, `int64`, and so on. This module also supports the conversion between the types `int`, `Z`, and `nat`.

Applicative finite maps are the main data structure used in the memory state and the global/local environment descriptions. There are two basic types, a `Tree` and a `Map`, from which a number of maps and trees can be derived. The difference between the two is: for `Tree` the result of the $\langle get \rangle$ operation is an option type: if there is no data associated with the key, `None` is returned. For type `Map`, $\langle get \rangle$ always returns a data. If there is no data associated, a default value will be returned, which is given at initialization time. These two data structures are based on the abstract signature radix-2 search tree. And the derived trees and maps are named by their keys which can be integer or positive. The `Tree` is used to define the global and the local environments, which gather memory information, and map the reference identifier to data information. Since the environment corresponds to a memory contents, no information can be obtained if a nonexistent address is given. On the contrary, the memory contents is represented by a `Map` indexed by an integer. If a block in memory has not been allocated, it should return a default value `Undefined` by any visit.

2.3.2 CompCert C semantics

`CompCert` C is a large subset of C language. Here are some limitations in this subset.

- Types: most of the types in C90 (1) are supported, except the following points.
  1. Unprototyped function type ($intf()$) and function type with variable number of arguments ($intf(...)$). But it is possible to declare (not define) an external function of the latter.
  2. A structure can not have an unknown sized array type as the last element. The size information must be known.
- Wide char and wide string.
- Type cast does not support pointer to float.
- Specify bit fields in unions are not supported.
- For the switch statement, `case` and `default` must appear. And the `default` must occur at last.

$$\frac{\begin{array}{c} G, E \vdash rf\ M \xRightarrow{t1} rf', M1 \qquad G, E \vdash rarg^*\ M1 \xRightarrow{t2} rarg'^*, M2 \\ G, E \vdash M2\ rf' \Rightarrow vf \qquad \texttt{find\_funct}\ (G, vf)\ =\ \lfloor fd \rfloor \\ \vdash M2\ fd\ varg^* \xRightarrow{t3} vres, M3 \end{array}}{G, E \vdash M\ \langle \texttt{Call} \rangle \xRightarrow{t1**t2**t3} vres, M3} \tag{2.14}$$

$$\frac{\begin{array}{c} G, E \vdash l\ M \xRightarrow{t1} l', M1 \qquad G, E \vdash r\ M1 \xRightarrow{t2} r', M2 \\ G, E \vdash l'\ M2 \Rightarrow (b, ofs) \qquad G, E \vdash r'\ M2 \Rightarrow v \\ cast(v, typeof(l), typeof(r)) =\ \lfloor v' \rfloor \\ store(G,\ typeof(l),\ M2,\ (b, ofs),\ v) =\ \lfloor M3 \rfloor \end{array}}{G, E \vdash (l = r)\ M \xRightarrow{t1**t2**t3} v', M3} \tag{2.15}$$

Figure 2.4: Some rules for `CompCert` C operational semantics

- The only available external functions are `printf`, `malloc`, `free`, `__builtin_annot` and `__bultin_annot_val`. The other external functions can be declared but not implemented. One external function will generate an event trace. It says the result of the external function is computed by operating system, not the `CompCert` C code.
- Every program must have a `main` function declared.

`CompCert` provides two operational semantics for `CompCert` C: one is non-deterministic, in small-step style; the other is detailed, in big-step style. In our case, the big-step semantics is enough for correctness proofs

The formal operational semantics is described as a transition system on memory states written as follows:

$$G, E\ \vdash\ \langle \text{expression} \rangle,\ M\ \xRightarrow{t}\ v,\ M'.$$

Here G represents the global environment of the whole program; E is the local environment; M and M' are memory states and t is a trace of I/O events; v is a returned value.

In `CompCert` C, expressions can be categorized into 15 cases, 13 of them are used in our correctness proofs. Some of them are similar to the ones for `Clight` and are already listed in `CompCert` papers (37). Inference rules that are different from `Clight` are presented in Fig. 2.4.

The first rule in Fig 2.4 is for evaluating a function call. The evaluation is quite different from the rule for `Clight`. Not only that `Clight` expressions are side-effect free, but `CompCert` C separates memory state transformation from evaluating simple expressions in order to preserve memory state. A function call can be evaluated in three steps: evaluating the function referenced by identifier `rf` to get where it is stored; evaluating the function arguments `rargs` to get their values; finding the function definition `fd` in the environment; then evaluating the function call using `eval_funcall`.

The second rule in Fig 2.4 is the evaluation of an assignment. In `Clight`, an assignment is not an expression but a statement because `Clight` expressions are pure.

In `Simlight`, the interpretation uses a subset of C features which is as simple as possible. This is not only to satisfy to `CompCert` C restrictions, but also to avoid ambiguous situations where an expression could have different behaviours. This way, the bigstep semantics of `CompCert` C is sufficient. However, some features outside of `CompCert` C occur in the current version of `Simlight`: external functions, which are used in many places to perform I/O subsystem communications. Currently, those external functions are represented by axioms. As a future improvement, it will be better to use internal functions instead.

Chapter 3

Formal model of ARMv6

In the beginning of this chapter, we present a short introduction of the ARMv6 reference manual, with an emphasis on the parts we have to specify in our model. Next we present our formal Coq model of ARMv6: the main types, how instructions are formalized, and the ARM instruction decoder.

Résumé

Ce chapitre commence par une introduction au manuel de référence de l'ARMv6, qui sert de point de départ de notre travail. Nous insistons plus particulièrement sur les parties que nous avons formalisé en Coq. Nous présentons ensuite notre modèle formel Coq de l'ARMv6 : les types principaux permettant de décrire l'état du processeur, la façon dont les instructions sont formalisées, et enfin le décodeur. Nous terminons par quelques remarques sur nos tentatives d'utilisation de ce modèle comme spécification exécutable.

3.1 The ARM reference manual

In order to certify the ARMv6 simulator, first we need to have a formal model that can be referred to. In an ideal world, the ARM company would provide a formal model of their processors, but it is not the case... In fact, the only basis we can depend upon to obtain an ARMv6 formal model is their reference manual (2). Similarly, `CompCert` project (35) designed their `CompCert` C language and Asm language according to the

informal ISO C 90 standard document (1) and relevant parts of the reference manuals for PowerPC, ARM, and IA32.

The ARMv6 reference manual is structured in four main parts, **CPU architecture**, **Memory and system architecture**, **Vector floating-point architecture** and **Debug architecture**. The useful contents for us to build our formal model is taken in the CPU architecture part. There is another important section at the end of the document, the **Glossary**, which gives the detailed explanation of key words appearing in the document using formulas and English.

The **CPU architecture** part introduces the ARM programmers' model, the ARM instruction set, the ARM addressing modes, and the Thumb instruction set. The contents of the programmers' model helps to formalize a *state* representing the structure of the ARMv6 processor. Most of the contents are written in English. The ARM processor state can be illustrated as in Figure 3.1 .

$$state_{proc} = \begin{cases} general\text{-}purpose\ registers \times \\ status\ registers \times \\ exceptions \times \\ processor\ modes \end{cases}$$

$$state_{scc} = registers \times memory$$

$$state_{ARMv6} = state_{proc} \times state_{scc}$$

Figure 3.1: ARM processor state

The ARM main processor contains thirty-one 32-bit general-purpose registers including the program counter, and six 32-bit status registers. A particularity of ARM architecture is that the program counter, register 15, can be used as any other general-purpose registers (e.g one can XOR the program counter with another register...) But it has many instruction-specific effects on instruction execution. If the program counter is used in a way that does not obey specified restrictions, the instruction will yield to an `UNPREDICTABLE` state. When `UNPREDICTABLE` state is reached, the instruction results cannot be relied upon, but the system will not halt or raise exception: `UNPREDICTABLE` is part of the system.

Access to the registers is decided by the current processor mode. The processor mode is encoded in 5 bits of the Current Program Status Register (CPSR), which is accessible in all processor modes (user mode, FIQ mode, IRQ mode, supervisor mode,

abort mode, undefined mode, system mode), see Figure 3.2. Other than CPSR, the other five status registers are the Saved Program Status Registers (SPSR) corresponding to each mode. When internal or external sources generate exceptions, the processor will react as follows. The processor status in CPSR will first be preserved into an SPSR according to the type of the exception. The processor mode is switched to the associated exception mode, the bits representing the processor mode in CPSR then change to the corresponding exception mode. Thus each type of exception proceeds under the specific exception mode. Eventually, the processor will return to the normal user mode and the CPSR will be restored from the saved value.

$$exn\_mode = fiq \mid irq \mid svc \mid abt \mid und$$

$$proc\_mode = usr \mid exc\_mode \mid sys$$

Figure 3.2: ARM processor modes

In our work, we only consider a simplified memory model without the memory management unit (MMU) described in part **Memory and system architecture**, in which only read and write functions are kept. In the ARM model, memory is controlled by the System Control Co-processor (CP15). So we describe the memory model as a part of System Control Co-processor (SCC), together with registers of SCC. The state of SCC and the state of the main processor are together to form the state of ARMv6.

The next part of **CPU architecture** is for the ARM instructions set. This is where we spent most of our efforts. For ARMv6 architecture, there are 147 kinds of instructions and five kinds of addressing modes; with the different combinations of 15 condition modes, 11 operands, and two post-operation modes; the combination of which represents tens of thousands of instruction instances to consider.

The reference manual describes the ARM instructions in a well-structured way by providing their syntax and usage. Each instruction is specified by:

- The instruction encoding table
- The instruction syntax
- The validation under different versions of ARM architecture
- The exceptions that may occur
- The pseudo-code explaining the instruction operation

– The usage or notes.

The description of instructions plays a very important role in our project. As an example, let us take the instruction `ADC` from the data-processing instruction set. This instruction performs an add with carry. It adds (with carry) the value of a register with either the value of another register or an immediate value. Its encoding table is shown in Figure 3.3 .

31 ... 28	27 26	25	24 ..21	20	19 ... 16	15 ... 12	11 0
cond	0 0	I	0 1 0 1	S	Rn	Rd	shifter_operand

Figure 3.3: ADC instruction encoding table

The bits [31 : 28] encode the conditions (`cond`), under which the instruction is going to be executed.

```
cond =  EQ | NE | CS | CC | MI | PL | VS | VC | HI |
        LS | GE | LT | GT | LE | AL
```

When the condition is not satisfied, the instruction has no effect on the processor state, acting like a *No-Op* instruction, and the program counter moves up to the next instruction. Most ARM instructions are conditional, only a small number can be executed unconditionally, although in practice many instructions bear the **always** code, which indicates that the instruction is always executed. The four bits of the condition code are related to the condition flags in the Status Register, so that an instruction can be executed only when the condition code matches.

The bits [24 : 21] form what is called the `opcode`, which specifies the operation. Here it contains the code for "add with carry". These bits are first checked by the decoder to recognize the instruction kind.

The `I` bit is an identifier which distinguishes the immediate shifter operand from the register-based shifter operand, and the `S` bit indicates whether the instruction updates the flags in CPSR.

`Rn` is the first source operand. According to the addressing mode encoded by bits `I`,7 and `4` (explained below in this section), the second operand is one of the following basic cases:

– An immediate operand, formed by rotating bits [7 : 0] with a even value decided by the four bits [11 : 8].

– A register operand, which refers to the bits [3 : 0].
– A shifted register operand, which is a shifted or rotated value of a register. The register is of bits [3 : 0], and the five types of shift is indicated by the bits [11 : 7].

However, because the ARM encoding is very dense, the same instruction with a special combination of the three bits `I`, `7`, and `4`, is no longer a data-processing instruction, but it becomes an extension of the Load/Store instruction...

In the assembly language description for each instruction, parameters wrapped by `< >` refer to corresponding bit fields in the encoding table. For example in Figure 3.4, the value of the parameters in the assembler syntax of `ADC` , `cond`, `S`, `Rd`, `Rn`, and `shifter_operand` must be encoded precisely by the bit fields from the `ADC` encoding table as Figure 3.3.

```
ADC {<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

Figure 3.4: ADC assembler syntax

The binary decoder in the simulator must extract the bit fields from the binary instruction and initialize C variables or data structures with the appropriate value of the parameters, which sometimes requires an additional operation like sign extension.

The C like code in Figure 3.5 is the piece of pseudo-code given in the document to describe what the instruction ADC does. It first checks if it is conditionally executed. If not, the execution will finish. If so, it assigns the result of adding the contents in register `Rn`, the value of `shifter_operand` and the carry (`C` flag in CPSR). Then, considering whether the `S` bit is set or not, the CPSR is updated or not. If the `S` bit is set and register `Rd` is the program counter, it means that the ARM processor is currently running under exception mode and the status in SPSR needs to be restored in CPSR. Before writing to the CPSR, a check is performed to determine if the current mode is an exception mode, because only such a mode possesses an SPSR. Otherwise the instruction returns `UNPREDICTABLE` . If the register `Rn` is some other general-purpose register and the `S` bit is set, CPSR has to be updated according to the result of the addition and the presence of a carry.

There are some pitfalls related to the different meaning of symbols in the pseudo-code. The same notation on different sides of '=' can be different. For example, let us consider the assignment "`Rd = Rn + shifter_operand + C Flag`". On the left-hand side of '=', the meaning of `Rd` is the address of `Rd` (the result will be assigned to

```
if ConditionPassed(cond) then
   Rd = Rn + shifter_operand + C Flag
   if S == 1 and Rd == R15 then
      if CurrentModeHasSPSR() then
         CPSR = SPSR
      else UNPREDICTABLE
   else if S == 1 then
      N Flag = Rd[31]
      Z Flag = if Rd == 0 then 1 else 0
      C Flag = CarryFrom(Rn + shifter_operand + C Flag)
      V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
```

Figure 3.5: ADC instruction operation Pseudo-code

address of `Rd`). On the other hand, the same `Rd` on the right-hand side, e.g., in "`N Flag = Rd[31]`", means the content of register `Rd` . More subtle, the meaning of `Rd` is different from the meaning of `Rn` – so such names are more like keywords than identifiers: on the right-hand side of an equation, `Rn` represents the *original* contents of `Rn` , whereas `Rd` represents the *current* contents of `Rd` . And when `Rd` and `Rn` happen to be the same register, the value of `Rn` on the right hand-side must stick to the original contents, not to the updated result.

As we explained for instruction `ADC` , the second operand is encoded by addressing mode 1 – data-processing operand. There are five groups of addressing mode:

- Addressing mode 1 – Data-processing operand
- Addressing mode 2 – Load and store word or unsigned byte
- Addressing mode 3 – Miscellaneous loads and stores
- Addressing mode 4 – Load and store multiple
- Addressing mode 5 – Load and store coprocessor.

Each of them contains several formats used to calculate the value used in the instruction operation. For example, in addressing mode 1, there are eleven formats to encode the `shift_operand`, and in addressing mode 2, there are nine. The reference manual gives for each format an encoding table and an assembler syntax, and its operation in pseudo-code, which are similar to the description of ARM instructions. With the usage and notes of each instruction, we are able to match the instruction with its

own addressing mode.

3.2 Formalization in Coq

We want the formal specification to be as close as possible to the reference manual. We also want it to be as simple as possible. In this way, our formal model can be reliable and simple enough to reason about.

Our formal model contains the basic library for integer representation and binary operations, a memory model, the main processor, the system control co-processor, the instruction set, the simulation loop, and a description of the initial configuration. The bit vector definition and operations reuse the integer module from `CompCert`, instantiated to 32-bit words, 4-bit words for register numbers and 30-bit integers for memory addresses.

The core part of the ARM processor is the ARM instruction set. Its formalization is a rather heavy piece of work. In particular, the pseudo-code of each instruction given by the ARM reference manual has to be formalized. As a result, we get a formal semantics for ARMv6 instructions.

According to the structure presented in Figure 3.1, we formalize a `Record` type `state` by composing the `state` of the main processor and the `state` of the system control coprocessor:

```
Record state : Type := mk_state {
  (* Current program status register *)
  cpsr : word;
  (* Saved program status registers *)
  spsr : exn_mode -> word;
  (* Registers *)
  reg : register -> word;
  (* Raised exceptions *)
  exns : list exception;
  (* Processor mode *)
  mode : proc_mode
}.
```

```
Record state : Type := mk_state {
```

```
  (* registers *)
  reg : regnum -> word;
  (* memory *)
  mem : address -> word
}.

Record state : Type := mk_state {
  (* Processor *)
  proc : Arm6_Proc.state;
  (* System control coprocessor *)
  scc : Arm6_SCC.state
}.
```

Considering the whole simulation system, we need another state representing not only the processor but also the execution status. We introduce a new type named `semstate` to distinguish it from the `state` for processors. The specification follows the monadic style (56) to represent calculations on the ARM processor states.

This style takes the sequentiality of transformations on the state into account. In the state monad, functions take a state as input and return a value combined with a new state. Beyond the `state`, two other pieces of information are handled: `loc`, which represents local variables of the operation, and `bo`, a Boolean indicating whether the program counter should be incremented or not; they are registered in the following record which is used for defining our monad.

```
Record semstate := mk_semstate { loc : local ;  bo : bool ; st : state }.

Inductive result {A} : Type :=
    | Ok (_ : A) (_ : semstate) | Ko (m : message) | Todo (m : message).

Definition semfun A := semstate -> @result A.
```

Note that in most cases, functions will return an *Ok* value. The value *Ko* is used for `UNPREDICTABLE` states and is implicitly propagated with our monadic constructors for exceptions. The value *Todo* is used in a similar way for unimplemented operations – currently, it is still the case for coprocessor instructions.

The simulation fetches the binary code at a given address; decodes it to corresponding assembly instruction; invokes the operation in library and executes it; and at last includes the computation of the address of the next instruction. The ARMv6 behavior semantics is described by functional rather than relational definitions. This means our specification is consistent and deterministic. The two main components of a processor simulator are then:

- The decoder, which decodes a given binary word, retrieves the name of an operation and its potential arguments in assembly code. In Section 3.2.2 we will explain how it is generated from the reference manual.
- The precise description of transitions is the operation of instruction. The definition contains operations on processor registers and memory; thereby, the processor state is changed. In the ARMv6 reference manual, these algorithms are written in a "pseudo-code" syntax which calls low-level primitives. For example, some code indicates setting a range of bits of a register by a given value. And some operations might lead to unspecified or forbidden results. In ARM processor, this is called `UNPREDICTABLE` . When the simulator meets these result, it then returns a `Ko` or `Todo` state with a message specific to the situation.

3.2.1 Running an instruction

Each instruction operation (O) from the reference manual, for example in Figure 3.5, gives a semi-formal description on how a instruction evaluates. Here we show how to specify it in by a corresponding Coq function (O_coq).

Taking the instruction `ADC` as an example, its formalization in Coq is showed in the following function `ADC_step`, which operates on the parameter `st` of type `state`. Other parameters are found by searching for unspecified variables in the pseudo-code. Not all variables are declared globally. Variables which are assigned during the execution are local variables except the output registers, for example, `Rd` . The body of the function is kept as close as possible to the pseudo-code by using notations, like `<st>`, as explained below:

```
(* A4.1.2 ADC *)
Definition ADC_step (S : bool) (cond : opcode) (d : regnum) (n : regnum)
                    (shifter_operand : word) : semfun _ := <s0>
```

```
if_then (ConditionPassed s0 cond)
  ([ <st> set_reg d (add (add (reg_content s0 n) shifter_operand)
                  ((cpsr s0)[Cbit]))
  ; If (andb (zeq S 1) (zeq d 15))
     then (<st> if_CurrentModeHasSPSR (fun em =>
                                        (<st> set_cpsr (spsr st em))))
     else         (if_then (zeq S 1)
        ([ <st> set_cpsr_bit Nbit ((reg_content st d)[n31])
        ; <st> set_cpsr_bit Zbit
                (if zeq (reg_content st d) 0
                 then repr 1
                 else repr 0)
        ; <st> set_cpsr_bit Cbit
                (CarryFrom_add3 (reg_content s0 n)
                                shifter_operand ((cpsr s0)[Cbit]))
        ; <st> set_cpsr_bit Vbit
                (OverflowFrom_add3 (reg_content s0 n)
                                   shifter_operand ((cpsr s0)[Cbit])) ])) ]).
```

For most of the ARMv6 instructions, executions are conditional. These conditionally executed instructions must first check if the current condition (argument `cond`) fits the required condition. Otherwise, the following operations will be skipped, and then go to the next instruction. The `S` bit argument indicates the instruction must update the status register `CPSR`. If the argument `Rd` refers to the program counter (R15), the updating of `CPSR` is going to preserve the value of `SPSR` when the current processor mode is one of the exception mode. If `Rd` is one of the other general-purpose register, updating of `CPSR` is done by updating the significant flags in `CPSR`. The values are calculated by operations on argument `Rn` which contains the first operand, and `shift_operand` which specifies the second operand.

Here we explain the notation `<st>`. It is the notation for function `_get_st`, a monadic function that provides access to the current state `st` at any place of the operation sequences:

```
Definition bind {A B} (m : semfun A) (f : A -> semfun B) : semfun B :=
  fun lbs0 =>
  match m lbs0 with
    | Ok a lbs1 => f a lbs1
    | Ko m => Ko m
```

```
    | Todo m => Todo m
  end.

Definition bind_s {A} fs B (m : semfun unit) (f : A -> semfun B) :
  semfun B :=
  bind m (fun _ lbs1 => f (fs lbs1) lbs1).

Definition _get_st {A} := bind_s st A (Ok tt).

Notation "'<' st '>' A" := (_get_st (fun st => A))
  (at level 200, A at level 100, st ident).
```

In general, every operation function terminates with `Ok` state. However, errors are implicitly propagated with our monadic constructors: `Ko` and `Todo`.

The other notations to keep the formalization well structured are the case statements `If`, `then`, `else`, and `if_then`, also the sequence statement denoted by brackets and semicolons:

```
Definition if_then_else {A} (c : bool) (f1 f2 : semfun A) : semfun A :=
  if c then f1 else f2.

Notation "'If' A 'then' B 'else' C" := (if_then_else A B C)
                                                  (at level 200).

Definition if_then (c : bool) (f : semfun unit) : semfun unit :=
  if_then_else c f (Ok tt).

Definition _set_bo b lbs := ok_semstate tt (loc lbs) b (st lbs).
Definition block (l : list (semfun unit)) : semfun unit :=
  let next_bo f1 f2 := next f1 (_get_bo f2) in
  List.fold_left (fun f1 f2 =>
    next_bo f1 (fun b1 =>
    next_bo f2 (fun b2 => _set_bo (andb b1 b2))) ) l (Ok tt).

Notation "[ a ; .. ; b ]" := (block (a :: .. (b :: nil) ..)).
```

3.2.2 Decoder

Now we consider the formalization of decoding instructions. An instruction encoding table, e.g. like in Figure 3.3, summarizes all possibilities for this instruction in 32-bits representation. All the others will be decoded into `UNPREDICTABLE` or undefined. Then we can build an ARM instruction decoder for the ARMv6 architecture using all of the instruction encoding tables.

The main body of the decoder is a big pattern matching program. Each constructor is represented by 32 bits, either implicit or explicit. The Coq code in Figure 3.6 shows a thumbnail of the formal decoder, for the constructor corresponding to `ADC` .

```
Definition decode_conditional (w : word) : decoder_result inst :=
  match w28_of_word w with
  ...
    (*4.1.2 - ADC*)
    | word28 0 0 I_ 0 1 0 1 S_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    =>
      decode_cond_mode decode_addr_mode1 w
        (fun add_mode condition =>
          ADC add_mode S_ condition (regnum_from_bit n12 w)
                                    (regnum_from_bit n16 w))
  ...
  end.
```

Figure 3.6: Formalized decoder of conditional executed instructions

The decoding of ARM instructions is difficult because some bit configurations are ambiguous at first sight: they could be interpreted as different kind of operations. Such ambiguities are solved in the reference manual, which specifies a precedence order between the interpretations. In our formalization, this precedence order is implemented by the order used for the different bit patterns, in the global pattern-matching construct. In Coq, the pattern matching is considered from top to bottom: a value belongs to the ith constructor if and only if it could not match any previous pattern; a pattern covered by previous patterns is considered as redundant by the Coq type checker.

The 147 instructions are first partitioned into two groups, conditional and unconditional instructions. For ARM instructions, the condition field `cond` [31 : 28] indicates

the conditional execution of ARM instruction. These instructions will be checked first by matching the first four bits with $0b1111$ representing an unconditional execution. The others are grouped by ARM instruction categories. Instructions belonging to the same category do not conflict with each other by the wild-card mechanism. We also define 5 levels for grouping conditional instructions.

- All multiply instructions without accumulator. They can be covered by similar multiply instruction with an accumulator. Instructions without accumulator fill the bits $[15 : 12]$ with $0b1111$, whereas instructions with an accumulator using them refer to the register for accumulator operand.
- Some instructions from ARMv5 architecture use the notion `SBO` or `SBZ` to express that the instruction bit is read as one/zero whatever the value of the bit is and it cannot be rewritten. These instructions need to be checked then, otherwise they could be hidden by some of the new ARMv6 instructions.
- A few load/store instructions work under the privileged mode. Two significant bits P and W are assigned to a special combination of values to indicate this kind of instructions. We have to put them in higher priority, before the similar instructions working for the other processor modes.
- Instructions load/store from memory with a format other than `word` have a shape similar to the load/store with `word`, but the four bits $[7 : 4]$ are used to refer to the load/store length – indicating whether it is a `half`, `double word`, or a `signed byte`.
- The last group contains all the operations with addressing modes. For decoding this kind of instructions, the decoder for addressing mode has to be called first. The addressing mode decoders are introduced below.

In Section 3.1 we mentioned that the ARM instruction set admits five kinds of addressing mode. They are used to encode the specific values appearing in the instruction pseudo-code. The encoding tables for addressing mode are in the same form as the ones for ARM instructions, the formalization of addressing mode decoders are similar to the instruction decoder. The following definition shows one clause of the decoder for the addressing mode 1 – Data-processing operands, to indicate that the `shift_operand` is calculated by an immediate logical shift left.

```
Definition decode_addr_mode1 (w : word) : decoder_result mode1:=
```

```
match w28_of_word w with
...
(*5.1.5 - Data processing operands - Logical shift left by immediate*)
| word28 0 0 0 _ _ _ _ S_ _ _ _ _ _ _ _ _ _ _ _ _ 0 0 0 _ _ _ _ =>
  DecInst _ (M1_Logical_shift_left_by_immediate
                                    (regnum_from_bit n0 w) w[n11#n7])
 ...
end.
```

3.3 Experimentations

Altogether, we get an executable formal model of ARMv6 architecture, which can be translated to OCaml code by extraction of Coq code. However, for the executable version of the formal simulator, we could not integrate this extracted OCaml code because the extraction mechanism translates a Coq pattern, which matches more than one terms, into many OCaml patterns, which mention all possibilities one by one.

More precisely, from the Coq model of ARMv6, it is possible to extract OCaml code and compile it to an executable simulator and perform some tests. The `arm-elf-gcc` compiler is already used in our group to compile C tests into ELF files to be used in `Simlight`. These tests could be translated to a Coq representation, then extracted to OCaml. Running a simple direct sum test takes around five minutes. A sorting program would then need one day to be completed.

Directly simulating the ARM in Coq would even be worse. However, execution speed is not a concern in formal proofs, as far as no heavy computations steps are involved.

Chapter 4

Simulation of ARMv6 in C

Here we introduce `Simlight`, our certification target. `Simlight` is a light version of the simulator `SimSoC`, which includes only the ARMv6 instruction set simulator with a simplified memory model.

We also give a brief description of `Simlight` 2, which includes several optimizations from `SimSoC` to obtain a higher simulation speed. The extension to the optimized version 2 will be discussed as future work in conclusion Chapter 9.

Résumé

Ce chapitre est consacré à `Simlight`, la cible que nous cherchons à certifier. `Simlight` est une version allégée du simulateur `SimSoC`, qui ne contient que le simulateur de jeu d'instructions, avec un moèle mémoire simplifié.

Nous donnons aussi une brève description de `Simlight` 2, qui intègre plusieurs optimisations utilisées dans `SimSoC` afin d'accélérer la simulation. L'extension de notre travail de certification à la version 2 optimisée est considérée en perspective dans la conclusion, au chapitre 9.

4.1 Simlight

Similarly to the Coq formal model, `Simlight` contains a data structure in C to represent the ARMv6 processor. As the structure of the processor did not change between ARMv5 and ARMv6, this data structure was copied from the previous version of `SimSoC` for ARMv5. It was designed to optimize execution, which makes it rather

```
struct SLv6_Processor {
  struct SLv6_MMU *mmu_ptr;
  struct SLv6_StatusRegister cpsr;
  struct SLv6_StatusRegister spsrs[5];
  struct SLv6_SystemCoproc cp15;
  size_t id;
  uint32_t user_regs[16];
  uint32_t fiq_regs[7];
  uint32_t irq_regs[2];
  uint32_t svc_regs[2];
  uint32_t abt_regs[2];
  uint32_t und_regs[2];
  uint32_t *pc; /* = &user_regs[15] */
  bool jump;
};
```

Figure 4.1: ARM Processor state in C

different from the formalization in Coq that keeps things as simple as possible, and as close as possible to the reference manual.

The C definition of the processor state is given in Figure 4.1.

Similarly, the data structure `SLv6_Processor` contains the most important components of the ARM processor: a pointer to the location of the structure representing the Memory Management Unit (MMU), a status register structure for CPSR, an array for the status register structure of SPSR, a structure for CP15 (SCC), a field for the processor id (useful when there is more than one core), six arrays for registers of each processor mode, one field for PC, and a boolean field `jump` which indicates whether the instruction modifies the PC or not.

For a better illustration, the C definition of the status register structure is given in Figure 4.2.

In order to gain high speed for the simulator, the processor type has been designed to have several redundant fields; for example, the PC field is a pointer to the 15th register in user register array. Indeed, the PC field is significant as it allows to judge whether the execution is branched or not, so that the running program can be split

```
struct SLv6_StatusRegister {
  bool N_flag; /* bit 31 */
  bool Z_flag;
  bool C_flag;
  bool V_flag; /* bit 28 */
  bool Q_flag; /* bit 27 */
  bool J_flag; /* bit 24 */
  bool GE0; /* bit 16 */
  bool GE1;
  bool GE2;
  bool GE3; /* bit 19 */
  bool E_flag; /* bit 9 */
  bool A_flag;
  bool I_flag;
  bool F_flag;
  bool T_flag; /* bit 5 */
  SLv6_Mode mode;
  uint32_t background; /* reserved bits */
};
```

Figure 4.2: ARM status register structure in C

into code blocks. And the PC field is referred to many times during execution. There is another important optimization method in `SimSoC` simulation 1.2. In order to optimize execution time, the data structures used to represent the processor state do not reflect the hardware structure. For example, whereas a bit is used in hardware to store a flag, a boolean variable is used to represent that bit in the C code. For example the 4 bits in the status register that indicate the traditional N, Z, C, V flags (Negative, Zero, Carry and oVerflow) for condition code, are represented by 4 booleans variables. Similarly, an array of status registers is used to represent the status in the different modes, indexed by the current mode. It means that the pseudo-instructions code that manipulate these data structures must be translated by appropriate C code that accesses the corresponding data,

As a result, the C code of the semantics functions has a structure close to the pseudo-code in the reference manual but with additional access functions to access the data in an optimized implementation. Below is the example of the ADC instruction:

```
/* ADC */
void ADC(struct SLv6_Processor *proc,
    const bool S,
    const SLv6_Condition cond,
    const uint8_t d,
    const uint8_t n,
    const uint32_t shifter_operand)
{
  const uint32_t old_Rn = reg(proc,n);
  const uint32_t old_CPSR = proc->cpsr;
  if (ConditionPassed(&proc->cpsr, cond)) {
    set_reg_or_pc(proc,d,((old_Rn + shifter_operand) + old_CPSR.C_flag));
    if (((S == 1) && (d == 15))) {
      if (CurrentModeHasSPSR(proc))
        copy_StatusRegister(&proc->cpsr, spsr(proc));
      else
        unpredictable();
    } else {
      if ((S == 1)) {
        proc->cpsr.N_flag = get_bit(reg(proc,d),31);
        proc->cpsr.Z_flag = ((reg(proc,d) == 0)? 1: 0);
        proc->cpsr.C_flag = CarryFrom_add3
```

```
                              (old_Rn,shifter_operand,old_CPSR.C_flag);
        proc->cpsr.V_flag = OverflowFrom_add3
                              (old_Rn,shifter_operand,old_CPSR.C_flag);
      }
    }
  }
}
```

As the C language accepted by Compcert is a subset of the full ISO C language, the generator has been constructed such that it only generates C code in the subset accepted by the compcert compiler.

Nonetheless it can be compiled with other C compilers such as GCC to obtain better performance. Even though in this case, the resulting machine code is not guaranteed to be correct (there are well known GCC optimization bugs...), at least the original C code has been proven by our technique to be conformant with the ARM semantics.

The ARM V6 code generator not only generates the semantics functions, it also generates the decoder of binary instructions supported in V6 architectures. This decoder is obtained by compiling the opcodes information. The generated decoder is probably not optimal in performance, but as `SimSoC` uses a cache to store the decoded instructions, the performance penalty is marginal.

4.2 Optimization on Simlight version 2

In this section, we introduce the second version of `Simlight`, that include optimizations to reduce simulation time. The optimization methods can be categorized as follows:

- *flattening* is used in `Simlight` version 2, in order to merge some instructions with their addressing mode. The result of *flattening* can improve the simulation speed. How to perform instructions flattening is introduced in Section 5.4.
- Partitioning the semantics function into hot and cold ones. C compilers now supports the *hot* and *cold* attributes on functions. When a function is declared hot, the compiler generates code that minimizes execution time. When it is cold, it minimizes code size. It also generates directives for the linker to group the hot and cold functions together to increase program locality. The temperature information (i.e. hot or cold) is obtained by running the program on sample

input to generate profiling data, such as obtained with the GPROF profiling tool. Based on profiling data, the `SimSoC` generator can partition the functions between cold and hot in order to benefit from these compiler optimizations.

– Specialize the instruction of boolean parameter values.

– Remove the instructions about coprocessor because the coprocessor is not supported yet. It can save time in encoding and decoding phases.

Chapter 5

Designing the generation chain

An important part of the specification of the ARMv6 instruction set (the behavior of 147 instructions and 24 addressing modes) is defined in the ARMv6 reference manual using a pseudo-code notation. As explained in the previous chapters, the Coq formal model of ARM instructions and their C representation for `Simlight` are systematically derived from this pseudo-code. Most instructions are described using more than 10 lines of pseudo-code. Manually translating them one by one would be a repetitive and error-prone task. We then built a automatic generator, which takes the pseudo-code of instructions as input and returns their representations in Coq and in C. The corresponding generation chain is presented in this chapter.

Résumé

Une partie importante de la spécification du jeu d'instructions de l'ARMv6 (le comportement des 147 instructions et des 24 modes d'adressage) est définie dans le manuel de référence de l'ARMv6 au moyen d'une notation en pseudo-code. Ainsi qu'on l'a expliqué dans les chapitres précédents, le modèle formel Coq des instructions ARM et leur représentation en C pour `Simlight` sont systématiquement dérivés à partir de ce pseudo-code. Pour la plupart des instructions, la description en pseudo-code fait plus de 10 lignes. Leur traduction à la main serait une tâche répétitive et sujette à erreurs. Nous avons donc construit un générateur automatique, prenant en entrée le pseudo-code des instructions et retournant leur réprésentation en Coq et en C, et la chaîne de génération correspondante fait l'objet de ce chapitre.

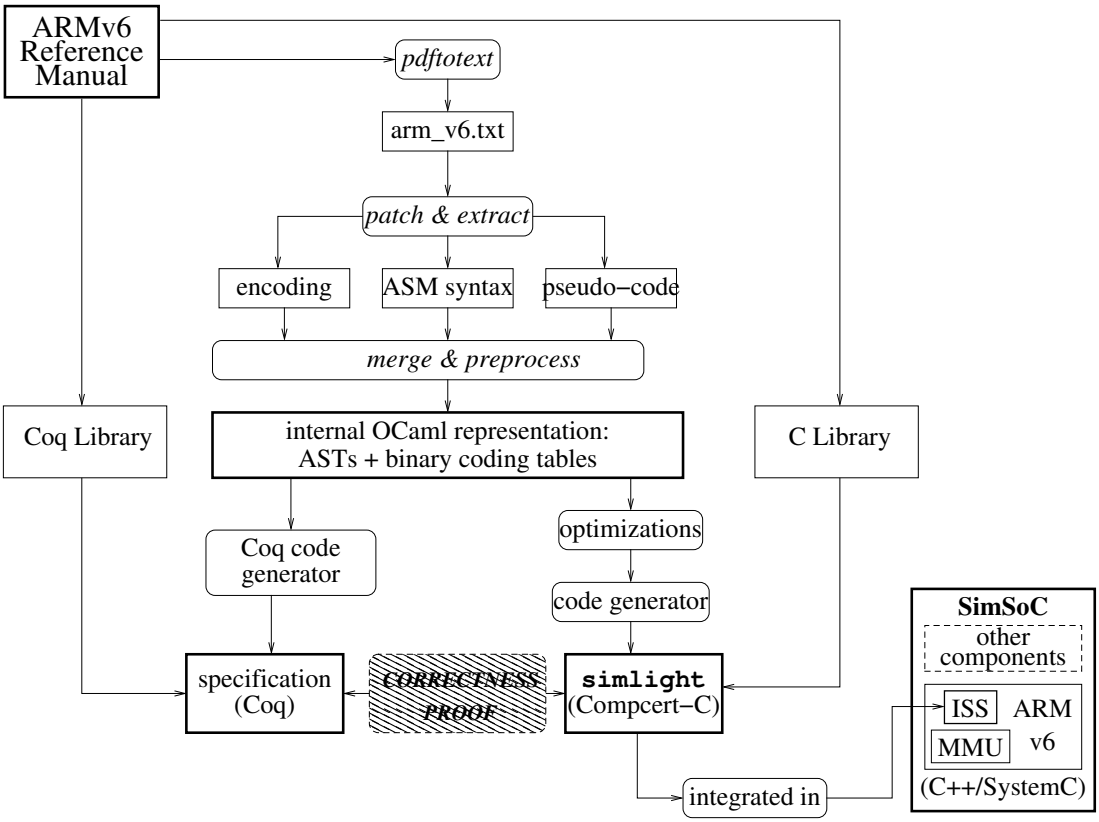

Figure 5.1: Overall generation chain

5.1 Architecture

The overall architecture of the automatic generator is given in figure 5.1.

The figure shows there are three data flows coming from manual. Two are going to be formalized manually; the other one part is going be interpreted and merged automatically.

More specifically, we can see the data flow from ARMv6 Reference Manual to the Coq model and to the simulation code. Some patches are needed from the textual version of the reference manual because the latter contains some minor bugs (see below).

Three kinds of information are extracted for each ARM operation: its binary encoding format, the corresponding assembly syntax, and its body, which is an algorithm operating on various data structures representing the state of an ARM: registers, memory, etc., according to the fields of the operation considered. This algorithm may call general purpose functions defined elsewhere in the manual, for which we provide a

CompCert C library to be used by the simulator and a Coq library defining their semantics. The latter relies on Integers.v and Coqlib.v from the CompCert library which allows us, for instance, to manipulate 32-bits representations of words. The result is a set of abstract syntax trees (ASTs) and binary coding tables. These ASTs follow the structure of the (not formally defined) pseudo-code.

In the end, three files are generated: a Coq file specifying the behavior of all operations (using the aforementioned Coq library), a CompCert C file to be linked with other components of SimSoC (each instruction can also be executed in stand-alone mode, for test purposes for instance) and a Coq files representing each instructions in CompCert C AST to be used for correctness proof.

5.2 Analysis of the ARM reference manual

The whole process starts with the ARMv6 reference manual ARM DDI 0100I (2). The relevant chapters for us are:

- Programmer's Model introduces the main features in ARMv6 architecture, the data types, registers, exceptions, etc;
- The ARM Instruction Set explains the instruction encoding in general and puts the instructions in categories;
- ARM Instructions lists all the ARM instructions in ARMv6 architecture in alphabetical order and ARM Addressing modes gives all the five kinds of addressing modes;
- Glossary gives all the definitions of key words in ARMv6. We use it as a reference to define manually the common functions.

There are 147 ARM instructions in the ARMv6 architecture. For each instruction, the manual gives its encoding table, its syntax, a piece of pseudo-code explaining its own operation, its exceptions, usage, and notes. Except the semi-formal pseudo-code, everything else is written in nature language.

The first step is extraction and patching. We extract three files from the reference manual: a 2100 lines file containing the pseudo-code, a 800 lines file containing the binary encoding tables, and a 500 lines file containing the ASM syntax. Other than these three extracted files, there are still useful information left in the document which cannot be automatically extracted. This is the case for the arithmetic functions given

in chapter `Glossary`, and for the validity constraints information required by the decoder generator. The corresponding information is manually translated into a 300 lines OCaml file.

Before extraction, a patch is necessary for the main text file. This patch is obtained from reading the manual or feedback from the generation result. The patch fixes the mistakes in the original document, such as misspelling function names, unclosed parenthesis, missing line, etc. Most of these bugs were found by running the generator or testing the generated simulator. The differences are kept in a diff file, so that they could be submitted to the ARM company and confirmed.

Then each extracted file is parsed with the corresponding parser. The one to parse pseudo-code is more complicated. Two preliminary phases solve issues related to line breaks and indentation, given that indentation defines the blocks in Python-like way.

Then, a classical lexer parser combination builds the abstract syntax trees (ASTs). We have built our own ASTs for intermediate representation which contains the elements representing both instructions and their addressing mode.

5.3 Intermediate representation

The abstract syntax of the intermediate representation expressions is given in Figure 5.2. The corresponding OCaml definition is an inductive data type. The type of expression supports numbers in different bases, conditional expressions, function calls, binary operations, ranges (e.g. Rn[31:0] indicate the range of bits 0 to 31 of register Rn), and the particular expression of ARM registers (e.g. `CPSR`, `SPSR`, and `Reg`), memory and coprocessor. Additionally, two key words are included: `Unaffected` which indicates the item is not changed by an operation, and `Unpredictable_exp` which represents an unreliable instructions result. The evaluation of expression `Unpredicatable_exp` and `Coproc_exp` can bring side-effect.

Figure 5.3 defines the abstract syntax of instruction statements, which is defined in type *inst*. The C-style structural statements are supported: blocks, assignments, conditional statements, loops (while loop and for loop), assert, case, and return. Special function calls related to processor and coprocessor are presented individually. Within statements, `Unpredictable` appears again. In pseudo-code, `UNPREDICTABLE` is used as

```
exp ::= num
      | bin
      | hex
      | float
      | if exp then exp else exp
      | fun (exp list)
      | exp binop exp
      | CPSR
      | SPSR mode option
      | Reg mode option
      | var
      | exp of range
      | Unaffected
      | Unpredictable_exp
      | Memory size
      | Coproc_exp exp list
mode ::= Fiq | Irq | Svc | Abt | Und | Usr | Sys
range ::= bit | flag | index
size ::= byte | half | word
```

Figure 5.2: The abstract syntax of intermediate representation expressions

```
inst ::= block inst list
       | let fun (args) = inst list
       | Unpredictable
       | exp = exp
       | if (exp) inst inst option
       | Proc_function exp list
       | while (exp) inst
       | assert exp
       | for (string) inst
       | Coproc_function exp list
       | case (exp) inst list
       | return exp
```

Figure 5.3: The abstract syntax of intermediate representation statements

the expression of right value in the assignment (e.g. `data = UNPREDICTABLE`), or as the statement of call to the function (e.g. `if...then...else UNPREDICTABLE`).

5.4 Code generation

On the formal specification side (left side of the generation chain in Figure 5.1), we directly use the ASTs for generating Coq code.

For the generation of C source code, we can make an easy optimization to generated the second version of `Simlight`, in order to improve the simulation speed: as we mentioned in Section 4.2, *flattening* is one way of improving the simulation performance. We *flatten* some instructions with their addressing mode When an instruction A can be used in an addressing mode B, the generation provides a combined instruction AB. This simple optimization can make the generation steps shorter and the generated code faster. And after flattening, the notions of addressing modes disappear.

This flattening step is achieved by four operations:

- Inlining the addressing mode to instruction operation code;
- Appending the validity constraint information;
- Merging the encoding table of the instruction and addressing mode case (example in figure 5.4)
- Merging the ASM syntax of the instruction and addressing mode case

(a) binary encoding of the `ADC` *instruction*

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 0
`cond`	`0 0`	`I`	`0 1 0 1`	`S`	`Rn`	`Rd`	`shifter_operand`

(b) binary encoding of the "logical shift left by immediate" operand

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 7	6 ... 4	3 ... 0
`cond`	`0 0`	`0`	`opcode`	`S`	`Rn`	`Rd`	`shift_imm`	`0 0 0`	`Rm`

(a+b) resulting binary encoding of the flattened instruction

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 7	6 ... 4	3 ... 0
`cond`	`0 0`	`0`	`0 1 0 1`	`S`	`Rn`	`Rd`	`shift_imm`	`0 0 0`	`Rm`

Figure 5.4: Flattening the ADC instruction with the shift left by immediate operand

There are some specific points for the pre-processing phase:

- We can have a *base register write-back* specification, saying that the base register which is used in address calculation will be modified. We have this case when `Rd == Rn`. The result is `UNPREDICTABLE` if the base register is PC. The base register write-back is disabled in M2, M3, M4 addressing modes.
- Some functions are reshaped depending on the number of arguments and the operation performed on them. For example, `CarryFrom(a + b)` is replaced by `CarryFrom_add2(a, b)`, which indicates that the carry is calculated from the "add" of two arguments.
- Some *if* or nested *if* expressions concern occur when there is at least one `UNPREDICTABLE` in the branches. They are merged by pre-processing in order to remove repetitive branches, so that we get at most one `UNPREDICTABLE` in a then-branch.

5.5 Formats for C code

To detail the generation of C implementations, we present a new Figure 5.5. In the first line, the C source code is generated directly from the optimized intermediate representation, and then go through the C parser provided by `CompCert` and a C to `CompCert` C interpretation. Both instructions and library are parsed into `CompCert` C ASTs. This result is integrated into the ARM simulator in SimSoC. The second line translates the intermediate representation AST into `CompCert` C AST, and then pretty print into Coq representation and C code too. From the AST translation, only the instructions are obtained. The result in Coq representation is used in the correctness

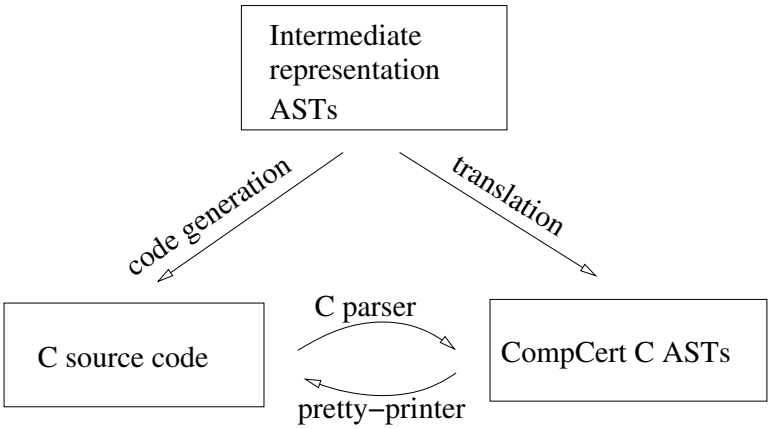

Figure 5.5: Generating C code

proofs. By comparison, the pretty printed C code and the `CompCert` code obtained from the C parser are identical to each other. Parsing to `CompCert` C does not lose any information, which means the subset of C, `CompCert` C, is large enough to be fulfill the requirement of ARM instructions.

Proofs are to be performed on `CompCert` C ASTs, so the more direct way such ASTs are obtained, the better, is in order to avoid possible mistaken auxiliary tranformations as far as we can (no proofs have been performed on parsers and pretty-printers for `CompCert` C). But it makes sense only for automatically generated C programs: writing ASTs by hand would be much too heavy and tedious. Therefore, we have two cases to consider. C libraries are written in textual format and parsed by the `CompCert` C front-end, while C instructions automatically derived from the pseudo-code are basically `CompCert` C ASTs and pretty-printed for a manual double-check in a readable form. As a result, for proofs, the `CompCert` C parser is in the trusted code base (TCB) for libraries, not for automatically derived code. If we execute the corresponding programs using the `CompCert` C compiler, the TCB is the same. If we execute the corresponding programs using another compiler, the TCB includes the pretty-printer and this compiler as well.

As a final remark on the reliability of the `CompCert` C parser and the pretty-printer, we also checked that, for all the generated code, parsing then pretty-printing yields the original code.

5.6 Mistakes in the ARM reference manual

While building the generators described in this chapter, we discovered several bugs in reference manual.

- Important lines were missing in instructions pseudo-code. In the operation of many conditional instructions, the condition checking was ignored. This leads to a fatal error when the execution condition is not satisfied. Also for some of load/store instruction, reading the base `address` is missing, which should be the content of register `Rn`. Without initialization, it is impossible to give `address` a value to start with.
- The case sensitivity gave the same spelling different meanings. For example, in the formal model, the binary operation *and* applies to type Boolean, but operation *AND* in capital is of type $word \rightarrow word \rightarrow word$. Mixing two of them will lead to a type mismatch.
- Information was lacking in keywords. For example, in general, `SignExtend` propagates the sign bit of its argument to 32 bits, but for instruction `BLX(1)`, `SignExtend` is for the 24-bit signed to 30 bits.
- Mismatched parenthesis.
- Wrong order of expressions in some operations.
- In assembly syntax, the expression of register content `Rx` had to be replaced by `<Rx>`.

These bugs have been reported to ARM group. The feedback was that all these bugs are fixed in ARMv7 reference manual.

Chapter 6

Correctness proofs

In this chapter we introduce the correctness proofs we have performed for the ARMv6 instruction set simulator `Simlight` by using the operational semantics of `CompCert` C. This work can be also considered as a significant experiment on proving C programs by using a formalized operational semantics of C.

Résumé

Ce chapitre est consacré aux preuves de correction que nous avons effectuées pour `Simlight`, le simulateur de jeu d'instructions de l'ARMv6 de notre projet, en utilisant la sémantique opérationnelle de `CompCert` C. Ce travail peut également être considéré comme une expérience significative de preuves de programmes C selon une approche basée sur la sémantique opérationnelle.

Essentiellement, nous avons à établir qu'un programme C représentant l'ARMv6 se comporte conformément au modèle Coq attendu, qui est un système de transitions sur un état abstrait directment défini en Coq. Le programme C, via la sémantique opérationnelle définie dans `CompCert`, est lui même modélisé par un système de transitions sur un état en un sens plus concret, qui est un modèle de la mémoire C (telle qu'elle est formalisée dans `CompCert`), habitée par des structures de données indiquées dans le programme `Simlight`. Bien que le programme C et le modèle Coq soient dérivés à partir des mêmes données du manuel de référence, et que la chaîne de génération de ces deux objets soit en partie partagée, on voit que ces objets sont de nature très différente. Le modèle Coq abstrait reste aussi simple que possible de façon à respecter

visiblement ce qui est énoncé dans le manuel de référence. En revanche, l'état concret pour `Simlight` prend non seulement en compte le modèle mémoire de `CompCert` C, mais des structures de données C complexifiées par un souci d'optimisation.

Afin de comparer le comportement du système de transition abstrait dans le modèle Coq et celui du système de transition concret correspondant à `Simlight`, nous commençons par définir une projection de l'état concret verts l'état abstrait. Nous pouvons alors énoncer, pour chaque instruction ARM, un théorème principal schématise en figure 6.1 (une version plus exacte est donnée plus loin en figure 6.2).

Le preuves s'effectuent alors en itérant l'analyse des hypothèses représentant des transitions entre états mémoire concrets, selon une relation appropriée de la sémantique opérationnelle à grand pas de `CompCert` C. La transition correspondant dans le modèle abstrait est représentée plus simplement par calcul, car dans le modèle Coq de l'ARMv6, les instructions sont représentées par des fonctions.

6.1 General idea

For the ARMv6 Instruction Set Simulator `Simlight`, we have to compare a Coq model with a C implementation (see Section 5.1).

In order to formally reason on the correctness of the second with relation to the first in the Coq setting, we need a formal model in Coq of the C implementation. It is provided by `CompCert`, which defines a operational semantics of C formalized in Coq. The two Coq models to be compared are state transition systems.

Note that a large part of these two models is automatically derived from the same source, that is, an AST representation of the pseudo-code for instructions taken in the ARMv6 manual. However, even for this part, it is far from obvious that the two models behave the same. They are actually quite different from each other.

Basically, the Coq specification follows exactly ARMv6 reference manual, and keeps everything as simple as possible. whereas the C program has more objectives to achieve because it is aimed to be a high speed simulator. In particular, states in the model of the C implementation are much more complex not only because the memory model defined in `CompCert` is taken into account, but also because of optimizations and design decisions in `Simlight` targetting efficiency. In more detail:

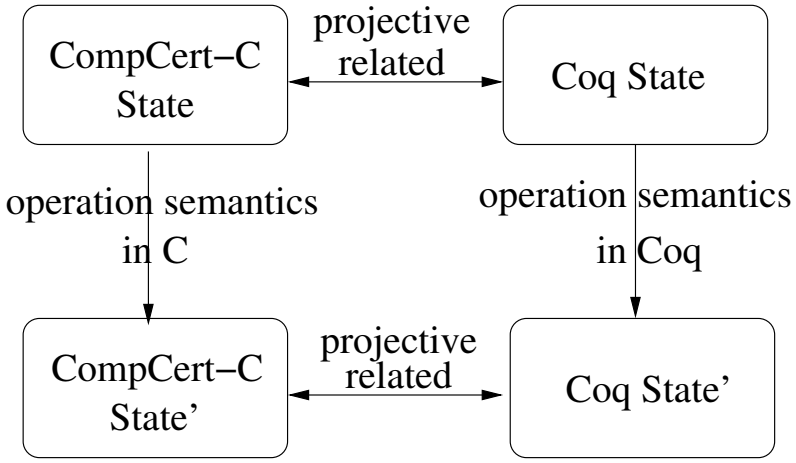

Figure 6.1: Main theorem for a given ARM instruction

- The C implementation uses a big *struct* to express the ARM processor state. The model of the state is a complex Coq record type, including not only data fields but also proofs to guaranteed access permission, next block pointer, etc. This is detailed in Section 6.3.
- In the Coq specification, transitions are defined in a functional style, whereas in the model of the C implementation, a relational style is used. In general, the relational style is more flexible but functional definitions have some advantages: reasoning steps can be replaced by computations; existence and unicity of the result are automatically ensured. However, the functional style is not always convenient or even possible. It is the case here, where the transitions defined by the C implementation are relations which happen to be functions. This comes first from the operational semantics, which needs to be relation for the sake of generality. Furthermore in our case, the kind of record type mentioned in the previous item is too complex to execute calculation with it, so it is more convenient to describe the state transformation for memory with a relation.
- The two semantics operates on very different states. For the Coq specification, reading or changing the value of the processor state or other related variables is easy to express. In the model of the program, the state is based on a complex memory model and load and store functions are used for read/write operations

For a given operation, we state and proof a main theorem which can be displayed by the diagram in Figure 6.1. On both sides, the execution of an instruction is de-

scribed by a state transition. For the two ISS representations, "State" refers to the full description of the system. We start from a C memory state corresponding to a more abstract state described by the Coq specification. This correspondance is expressed by a projection relating the two models of the state. Then, executing the same instruction on two sides will produce a pair of new processor states which are related by the same correspondance. Informally, executing the same instruction on a pair equivalent states will produce a pair of equivalent states.

6.2 The ARMv6 model in `CompCert` C

A `CompCert` C program a list of functions, including the program entry point called `main`, with global variables as parameters. The transformation from pseudo-code AST to `CompCert` C AST produces a standalone program for each ARMv6 instruction. Then each has its own correctness proof separately. In the generated `CompCert` C file, program contains only one function which is the instruction operation. Other invoked functions are not included because the instruction pseudo-code AST has nothing but a reference name. Their bodies are then manually included.

Every function is composed by its return type, function parameters, local variables, and the function body. The function body is a sequence of statements made of expressions. In `CompCert` ASTs, constructs are very detailed. Each expression and each statement is annotated with its own type. In a program, the same type may appear several times. In the raw output of an AST, large and repeated expressions for types occur everywhere, making `CompCert` ASTs much more verbose and space consuming than necessary and very hard to read. In order to solve this issue and, more generally, get a readable code, the pretty printer for ASTs introduces auxiliary names for types – common subtypes are then shared – and also uses special notations for most constructs expression. The implementation of this part was contributed by Frédéric Blanqui and Frédéric Tuong.

As a result, the code for `CompCert` C ASTs of instructions becomes reasonably readable, as illustrated on the following example (the instruction `BL`, "Branch and Link").

```
Definition fun_internal_B :=
  {|
```

```
    fn_return := void;
    fn_params := [
      proc -: `*` typ_SLv6_Processor;
      L -: int8;
      cond -: int32;
      signed_immed_24 -: uint32];
    fn_vars := [];
    fn_body :=
    `if (call (ConditionPassed`:T1) E[&((`*(proc`:T2)`:T3)|cpsr`:T4)`:T5;cond`:T6] T7)
    `then `if ((L`:T7)==(#1`:T6)`:T6)
    `then (call (set_reg`:T8)
            E[proc`:T2; #14`:T6;
              (call (address_of_next_instruction`:T9) E[proc`:T2] T10)]
            T11)
    `else skip;;
    (call (set_pc_raw`:T12)
      E[proc`:T2;
        (call (reg`:T13) E[proc`:T2; #15`:T6] T10)+
        ((call (SignExtend_30`:T14) E[signed_immed_24`:T10] T10)<<(#2`:T6)`:T10)`:T10]
      T11)
    `else skip
  |}.
```

The textual version of this CompCert C code would be:

```
void B(struct SLv6_Processor *proc,
    const bool L,
    const SLv6_Condition cond,
    const uint32_t signed_immed_24)
{
  if (ConditionPassed(&proc->cpsr, cond)) {
    if ((L == 1))
      set_reg(proc,14,address_of_next_instruction(proc));
    set_pc_raw(proc,(reg(proc,15) + (SignExtend_30(signed_immed_24) << 2)));
  }
}
```

Another issue about the generated code is that the identifiers of variables, function names, and so on, have their own numerical values. These identifiers are important for referencing memory blocks. But for the same identifier, we may have different values in different CompCert C programs as in the current version, each instruction corresponds to one standalone program. This makes it difficult to share lemmas on common library

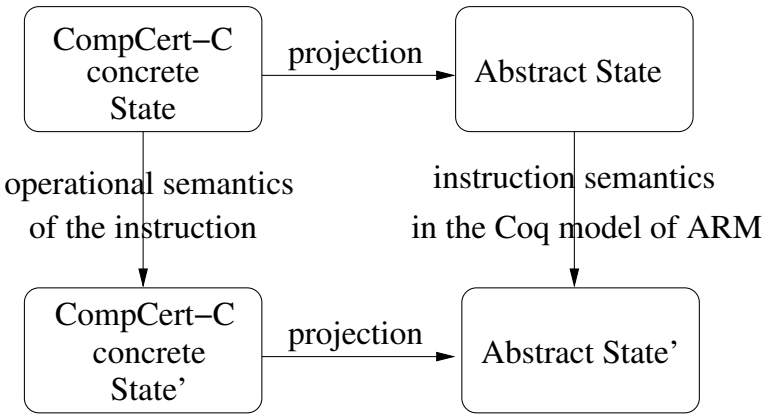

Figure 6.2: More accurate theorem statement for a given ARM instruction

functions used in several instructions. Our solution to this issue is discussed below in Section 6.4.3.

6.3 The projection

The state of the ARMv6 is defined in our Coq model in Figure 3.1. For convenience we will call this state the *abstract state*. On the other hand, the same state is represented in the Coq model of `Simlight` by the `CompCert` memory model applied to the data structure displayed in Figure 4.1. For convenience we will call this state the *concrete state*. In order to state correctness theorems on `Simlight`, we need to relate these two Coq models. To this effect, we define a projection from the concrete state to the abstract state.

Our theorems are then more accurately schematized by Figure 6.2 than in Figure 6.1 above.

Recall that our Coq model keeps everything as simple as possible and exactly corresponds to the ARMv6 reference manual, whereas the C representation is designed for high simulation speed. Moreover, additional complexity is introduced because a suitable memory model is required.

In the `CompCert` C model, variables are stored in the memory model. This `CompCert` C memory model is detailed enough to describe the real memory properties, but it is too complicated to use for computation. `CompCert` handles another auxiliary parameter *env*, the local environment. It maps each variable identifier to its location and its type,

and its value is stored in the associated memory block. The value associated to a C variable or a parameter of a C function is obtained by applying `load` to the suitable reference block in memory. However, this makes sense only after variable are allocated and initialized – these two operations are performed when a function is called, building a local environment `e` and an initialized memory state `m`. Similarly, our projection makes sense only at this stage, i.e., parameters representing the processor state are stored in memory. Our Coq model of ARMv6 is of course much simpler and computing the value of a component can be performed directly.

The abstract state of the processor in our Coq model is a record. It contains two records: one represents the main processor; the other has the system control co-processor(`SCC`) and a simple ARMv6 memory altogether. In the main processor record, the field `CPSR` (Current Program Status Register) is defined as a word; `SPSR` (Saved Program Status Register) is a word depending on current processor mode; `reg` maps the register to its value as a word; `exn` is a list of possible exceptions, which is not in use yet. `mode` is a numeration type for all processor modes. In `SCC`, there are only two elements, `reg` and `mem`: `reg` is the register owned by `SCC`, which maps the register identical number to its word value; `mem` is the ARMv6 memory model, which is a simple mapping from address to word value. We only have a trivial MMU (Memory Management Unit) for the moment.

The ARMv6 Processor data structure in C is given in Figure 4.1. It is a *struct* with thirteen fields, which in turn contains three *struct*: `SLv6_MMU`, `SLv6_StatusRegister` for `cpsr`, and `SLv6_SystemCoproc`, an array of *struct* `SLv6_StatusRegister` for `spsrs`, and six arrays for registers under each processor mode. The other three are: an identifier `id`, which is used when an embedded system has a multi-core architecture; a pointer `pc`, which points to the fifteenth of register array under user mode; and a boolean `jump` for expressing that the last instruction modifies the `pc`, to be cleared after each cycle. The *struct* `SLv6_StatusRegister` describes the status register, with bits represented as byte fields, plus one field to identify the current processor mode. The datatypes `CPSR` and `SPSRS` use this type. The difference is that not every processor mode has `SPSRS`. So an array for `SPSR` is used under every possible processor mode.

The top definition of the projection is shown below. Each sub projection refers to the link between an concrete element and its abstract version, in red color in Figure 6.3.

```
Definition proc_proj (m:Mem.mem) (e:env):Arm6_State.state:=
  Arm6_State.mk_state
    (Arm6_Proc.mk_state
       (cpsr_proj m e)
       (spsr_proj m e)
       (regs_proj m e)
       nil
       (mode_proj m e))
    (Arm6_SCC.mk_state
       (screg_proj m e)
       (mem_proj m e)).
```

For example, the projection for registers owned by the main processor is called `regs_proj`, which takes the C memory state `m` and the local environment `e` as arguments and return `register -> word`. The definition is as follow:

```
Definition regs_proj (m:Mem.mem) (e:env): register -> word :=
  let load_reg id n m e:=
    match find_reg m e id with
      | Some(Vptr b ofs)=>
        load_val (Mem.loadv Mint32 m (Vptr b (add ofs (repr n))))
      | _ =>Int.zero
    end in
    fun r =>
      match r with
        | R k => load_reg user_regs k m e
        | R_svc k _=> load_reg svc_regs k m e
        | R_abt k _=> load_reg abt_regs k m e
        | R_und k _=> load_reg und_regs k m e
        | R_irq k _=> load_reg irq_regs k m e
        | R_fiq k _=> load_reg fiq_regs k m e
      end.
```

Using the name of the register group as index to find the associated memory block, from which the value is loaded. Loading from memory state requires also the chunk information of its type, size and signedness, and the offset. In this case, the chunk of

register is `Mint32` which means it is 32-bit integer. If the corresponding register is not found in memory state, it returns zero. Initially, the value stored in register is zero.

According to the type of the argument on the right hand side of the projection, the definitions of projections are quite different. For example, the projection of a register given above performs a case analysis on a value of type `register`, whereas the projection of SPSR depends on the type of exception modes. We define a specific projection for each type. Coq is rich enough to allow us to define a general projection for all types of elements, using dependent types. However the gain in clarity of the specification is unclear, and it would anyway be just a wrapper around specific projections, so we did not build general protection for parameters. For improving readability of the statement, we even chose to define a projection relation for each instance. For example, the projection relation of register Rn is :

```
Definition rn_related (m:Mem.mem) (e:env) (rn:regnum):Prop :=
  reg_proj m e n = rn.
```

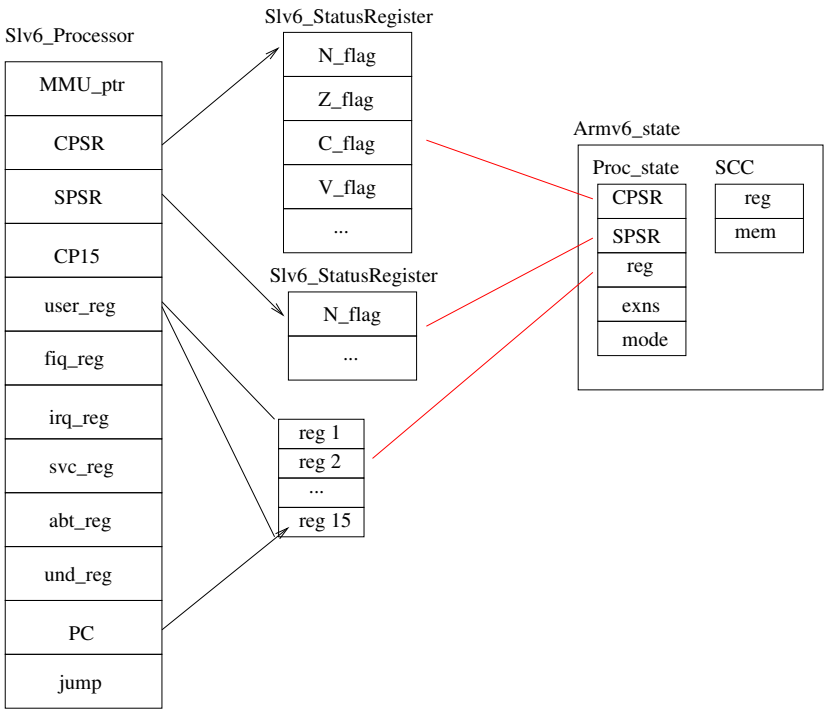

Figure 6.3: Projection

In another project in our group called CCCBIP, another way is used to express the projection. The `eval_expression` is reused to link the memory state and the variable value:

$$G, E \vdash \texttt{eval\_expression}\ (Ederef(Evalof(Evarx)))\ M\ \Rightarrow M,\ v.$$

The value of x in the formal model is v and the `CompCert` C expression $Ederef(Evalof(Evarx))$ is used to dereference the variable x from the memory contents. In this case the memory state remains the same because this evaluation only reads from memory. This technique can be used there because the type of values are very simple (integers). On the contrary, the types in `SimSoC-Cert` are much more complex: we have structure pointers inside structures, or arrays of structures inside structures, etc. Simply dereferencing with $Ederef$ as in CCCBIP would raise issues in our case. Manually writing such expression would become error-prone. Even more, this method results in more inverting tactics during proving, which makes the proof script harder to follow. And after inverting, the function `load_value_of_type` (or `deref_loc`), which loads from memory, will be added to hypotheses as the premise of evaluating the right value of expression `Evalof`. And it is just the same as the `load` function with premises on memory access mode predicate and type volatile judgment. But these premises would then be redundant with the existing hypotheses obtained during analyzing the evaluation of the expression where the corresponding variable is mentioned.

6.4 Proofs

6.4.1 Proofs for an ARM instruction

The correctness proof is based on the semantics of the formal model and the `CompCert` C representation. The semantics in the formalization is explained in Section 3.2.1. `CompCert` designed a semantics for `CompCert` C in both small-step and big-step. The big-step inductive type for evaluating expression is enough for our proof. The semantics is defined as a relation between an initial expression and an output expression after evaluation.

As mentioned before, the semantics of `CompCert` C considers two environments. The global environment *genv* maps global function identifiers, global variables identifiers to

their blocks in memory, and function pointers to a function definition body. The local environment *env* maps local variables of a function to their memory blocks reference. When the program starts its execution, *genv* is built. On the other hand, *env* is built when the associated function starts to allocate its variables.

To state the correctness theorem, we compare a `CompCert` C function corresponding to an ARM instruction with its formal definition in Coq. For such functions, it is enough to focus on the part of the concrete state which is defined by the local environment. We then consider a projection from the local environment to the abstract state defined as follows.

```
Inductive proc_state_related : Mem.mem -> env -> @result unit -> Prop :=
  | proc_state_related_ok :
      forall m e l b, proc_state_related m e
                        (Ok tt (mk_semstate l b (proc_proj m e)))
  | state_not_ok: forall e m mes, proc_state_related m e (Ko mes)
  | state_todo: forall e m mes, proc_state_related m e (Todo mes).
```

The shape of the main theorem of an instruction is then:

```
Theorem correctness_instr:
  ∀ e m0 m1 m2 mfin vargs st other_params out,
  alloc_variables empty_env m0 (fun_internal_B.(fn_params) ++
                                  fun_internal_B.(fn_vars)) e m1 ->
  bind_parameters e m1 fun_internal_B.(fn_params) vargs m2 ->
  (forall m ch b ofs, Mem.valid_access m ch b ofs Readable) ->
  proc_state_related m2 e (Ok tt (mk_semstate nil true st)) ->
  other_params_related m2 e other_params ->
  exec_stmt (Genv.globalenv prog_bl) e m2 fun_internal_B.(fn_body)
                                      Events.E0 mfin out ->
  proc_state_related mfin e (S.instr_step other_params
                               (mk_semstate nil true st)).
```

Let us explain it in more detail.

- In order to get the projection of the pair of original states, we need the following data: the initial memory state, the local environment, and the formal initial processor state. Recall that the projection is meaningful only after the C memory

state is well prepared for evaluating the current function body. In the abstract Coq model, we directly use the processor state `st`. But on the C side, the memory state must provide the contents of every parameter, especially the processor state. We also need to observe the modification of certain blocks of memory corresponding to local variables. Therefore, on `CompCert` C side, a memory state and a local environment is prepared using following two steps.

  - Allocating function variables: from an empty local environment, all function parameters and local variables ara allocated into the memory state `m0`, yielding a new memory state `m1` and the local environment `e`.
  - Initializing function parameters: using `bind_parameters` to initialize parameters with a list of argument values `vargs`, a new memory state `m2` is created.

- Now we have all elements for the projection to make sense are ready. As the most important parameter of instruction operation, the projection is first applied to `m2`, and we expect to get the initial abstract processor state `st`.
- The projection is also used on the other instruction parameters.
- Then the body of the function is executed. On the `CompCert` C side, this is performed using a call to `exec_stmt`, yeilding a new memory state `mfin`. On the abstract side, the new processor state is obtained using `instr_step`.
- Finally, we claim that the projection from the concrete state `mfin` should provide the latter abstract state. Note that all projections are performed using the same local environment `e`.

The proof is performed in a top-down manner. It follows the definition of the instruction, analyzing the expression step by step. The function body is split into statements and then into expressions.

When evaluating an expression, we search for two kinds of information. One is how the memory state changes on `CompCert` C side; the other is whether the results on the abstract and the concrete model are related by the projection. To this effect, we use six kind of lemmas.

1. *Evaluating a `CompCert` expression with no modification on the memory state.*
   Such a lemma only discusses the expression evaluation on `CompCert` C side, involving with the C memory state changing issue. Saying a memory state is not modified has two aspects: one is that the memory contents are not modified; the

other is that the memory access permission is not changed. For example, evaluating the binary expression $Sbit \ == \ 1$ returns an unchanged memory state.

$$\begin{aligned} &\text{if } \ G, E \ \vdash \texttt{eval\_binop}_c \ (Sbit \ == \ 1), \ M \ \stackrel{\varepsilon}{\Longrightarrow} \ vres, \ M' \\ &\text{then } \ M = M'. \end{aligned}$$

In Coq syntax, the relation in premise is expressed with `eval_binop`, a companion predicate of `exec_stmt` above, devoted to binary operations. In this lemma and the following, E is the local environment, G is the global environment and M is the memory state; ε is the empty event (`Events.E0` in Coq syntax); usually t is used to represent a series of system events; $vres$ is the result.

Here, $vres$ is not important. The evaluation is performed under environments G and E. Before evaluation, we are in memory state M. With no event occurring, we get the next memory state M'. The proof is easy. According to the definition of `eval_binop`, an internal memory state will be introduced.

$$\frac{G, E \ \vdash a_1, M \Rightarrow M' \quad G, E \ \vdash a_2, M' \Rightarrow M''}{G, E \ \vdash (a_1 \ binop \ a_2), M \Rightarrow \ M''}$$

Now, in our example, expression a_1 is the value of $Sbit$ and a_2 is the constant value 1. By inverting the hypothesis of type `eval_binop`, we obtain several new hypotheses, including on the evaluation of the two subexpressions and the introduction of an intermediate memory state M''. Evaluating them has no change on the C memory state. Then we have $M = M'' = M'$.

In more detail, from the `CompCert` C semantics definition, we know that, evaluation of an expression will change the memory state if the evaluation contains uses of `store_value_of_type` (in `CompCert` versions before 1.11), which stores the value in memory at a given block reference and memory chunk. In `CompCert`-1.11, the basic store function on memory is represented by an inductive type `assign_loc` instead of `store_value_of_type`. Since `CompCert` version 1.11 introduces volatile memory access, we have to determine whether the object type is volatile before storage, and also type size in addition of the access mode.

2. *Result of the evaluation of an expression with no modification on the memory.* Continuing the example above, we now discuss the result of evaluating the binary operation $Sbit \ == \ 1$ both in the abstract and the concrete model. At the end

of evaluation, a boolean value *true* or $false$ should be returned. in `CompCert` C model and Coq model, using the projection definition we introduced in 6.3.

if `Sbit_related` M `Sbit`,
and $G, E \vdash \texttt{eval\_rvalue\_binop}_c\ (Sbit\ ==\ 1), M \Rightarrow\ v$,
then $v = (Sbit\ ==\ 1)_{coq}$

Intuitively, if the projection corresponding to the parameter `sbit` in the C program yields the right information from the abstract state, then the evaluation will return the same value both in the abstract and in the concrete model. Here, the expression is a so-called "simple expression" that always terminates in a deterministic way, and preserves the memory state.

To evaluate the value of simple expressions, `CompCert` provides two other big-step relations `eval_simple_rvalue` and `eval_simple_lvalue` for evaluating respectively their left and right values. The rules have the following shape:

$$\frac{G, E \vdash a_1, M \Rightarrow v_1 \quad G, E \vdash a_2, M \Rightarrow v_2 \quad \texttt{sem\_binary\_operation}(op, v_1, v_2, M) = v}{G, E \vdash (a_1\ op\ a_2), M \Rightarrow v}$$

In order to evaluate the binary expression $a_1\ op\ a_2$, the sub-expressions a_1 and a_2 are first evaluated, and their respective results v_1 and v_2 are used to compute the final result v.

3. *Memory state changed by storage operation.*
   As mentioned before, evaluating some expressions such as `eval_assign` can modify the memory state. Then we need lemmas stating that corresponding variables in the abstract and in the concrete model will evolve consistently. For example, this is stated as follows for an assignment on register Rn. Here we use the projection relation `register_related`.

if `rn_related` $M\ rn$
and $G, E \vdash \texttt{eval\_assign}_c\ (rn := rx), M\ \Rightarrow\ M', v$
then `rn_related` $M'\ rn$

4. *Evaluating expressions with modification on the memory.*
   This is similar to the previous case.

5. *Internal function call.*
   Internal functions are described in an informal manner in the ARMv6 reference manual. No pseudo-code is available for them, which means that the corresponding library functions, both in the abstract Coq model and in `Simlight`, are written by hand. In order to get a suitable `CompCert` C AST to reason about, we use the parser provided in `CompCert`. When combining the simulation code of an instruction with the code of library functions, we need to take care of the memory allocation problem. In `CompCert` C representation, identifiers are unique positive numbers which indicate the memory block where corresponding variables are allocated. Currently, the extra identifiers introduced by library functions are added manually and assigned with fresh block numbers.

   $$\begin{aligned}
   &\text{if } \texttt{proc_state_related}\ M\ st \\
   &\text{and } G, E \vdash \texttt{eval_funcall}_c\ (copy_StatusRegister)_c, M \Rightarrow\ v,\ M' \\
   &\text{and } st' = (copy_StatusRegister)_{coq}\ st \\
   &\text{then } \texttt{proc_state_related}\ M'\ st'.
   \end{aligned}$$

   After an internal function is called, a new stack of blocks is allocated in memory. After the evaluation of the function is performed, these blocks will be freed. Unfortunately, this cannot bring the memory back to the previous state: the memory contents may stay the same, but the `nextblock` pointer will skip these just freed blocks and point to the followed block. For lemmas on evaluation of internal functions, we can observe the returned result on variables and compare it with the corresponding evaluation in the formal specification. For example, the lemma above is about the processor state after evaluating an internal function call `copy_StatusRegister` which reads the value of CPSR and then assigns it to SPSR. The evaluation of `copy_StatusRegister` should be protected by a check on the current processor mode. If it is neither system mode nor user mode, the function `copy_StatusRegister` can be called. Otherwise, `Simlight` will return "unpredictable" with an empty message.

Then we have to reason on the newly returned states, which should still be related by the projection. This step is easy to prove by calculation, simplifying on two representations of the processor state.

6. *External function call.*

   The `CompCert` C AST of an external function call contains the types of input arguments and of the returned value, and an empty body. `CompCert` provides the expected properties of a few built-in external functions such as `printf`, `malloc` and `free`. We proceed similarly for the external functions of `Simlight`.

   In `Simlight`, some functions are defined as external ones – something which is needed even is this simplified version of `SimSoC`. They could be changed into internal functions in the future but in the current version, they are left external.

   The general expected properties of an external call are as follows.

   – The call returns an result, which has to be related to the abstract.
   – The number of arguments must agree with the signature.
   – After the call, no memory blocks are invalidated.
   – The call does not increase the max access permission of any valid block.
   – The memory state can be modified only when the access permission of the call is the maximal.

   For `Simlight`, the result of an external call is written in a variable such as `vres` in the next example. A typical axiom for stating that the external function `ef_c` returns a result specified by the Coq expression `ef_coq` is:

```
Axiom res_extcall :
  forall m ef_c targs tres vargs t m' vres,
    eval_funcall m (External ef_c targs tres) vargs t m' vres ->
    vres = ef_coq.
```

6.4.2 Proof design

As usual, repetitive steps in proofs are dealt with using auxiliary lemmas and dedicated tactic definitions. In our case, most of them are related to the semantics of `CompCert` C. Indeed, since the abstract Coq model is defined in a functional style, many proof steps are just reductions using, e.g., `simpl` or `unfold`. In contrast, the execution of a C program is provided by a inductively defined relations, the operational

semantics. Decomposing this execution step by step amounts to perform so-called *inversions* on hypotheses relating concrete memory states according to the operational semantics. In practice, a large amount (several dozens) of inversions are performed, bringing serious issues on space-time consumption and maintenability. We studied a general solution to this problem, to be introduced in Chapter 7.

More specifically, back to the design of proofs, here are the main issues and how they are dealt with.

Getting a usable local environment. We often need to consider whether a variable exists in C memory or not, and to get the corresponding location in memory. To this effect, the concrete contents of the local environment `e` is required. To achieve this, inversions are systematically performed on `alloc_variables` hypotheses. Then `e` becomes a *closed* (and reduced) mapping indexed by variable identifiers (before, `e` is just a variable having the type of a mapping).

Finding a variable location in memory from its identifier. This is simply solved by applying the `get` operation provided by `CompCert` on the local environment `e`. This computation can actually be performed when `e` is closed.

Finding a function location in memory. Two kinds of functions exist in C. Internal functions are in the local environment `e`, whereas external functions are in the global environment `g`. When a function reference is met, a `get` operation is invoked on `e`, then on `g` in case of failure.

Evaluating memory states. `CompCert` C semantics operates on memory states. Observing them is essential, in particular to compare the concrete state with the expected abstract state. The memory state stays unchanged, except when a `store` occurs during evaluation. In the inductive relation `eval_expr`, this only happens for an assignment (`eval_assign`), an assignment with arithmetic operation (`eval_assignop`) or a post-increment operation (`eval_postincr`). Whatever the expression, it has to be analyzed and recursively decomposed in order to get closed (then usable) memory states. Again, this is performed using inversions on `eval_expr` hypotheses. The inductive type `eval_expr` is big and expressions in `Simlight` are complex, raising serious

issues with the Coq standard tactic `inversion`. We then decided to write our own inversion tactic. We go back to this in Section 7.1.

Analysing values in a memory block. The `CompCert` memory model includes four kinds of operations: `load`, `store`, `alloc`, and `free`. They operate on a memory chunk at a given address. For these four operations, several properties are provided. We use them to determine which block or memory chunk is affected by one of these operations, and which part of the memory is left unchanged.

6.4.3 Proofs for shared library functions

Every ARMv6 instruction contains one or more calls to internal or external functions. For the moment, the external functions are not taken into account, for a reason explained in 2.3.2. As mentioned in Section 6.2, functions called in an instruction need to be added manually. Most of these functions are used in different instructions. As the properties expected from them are always the same, we want to state and proof corresponding lemmas once for all.

One issue from the `CompCert` compiler is that identifiers (positive numbers indicating a location in memory) cannot be repeated:these memory locations are settled once a program is evaluated, and the global environment and local environment will be filled with allocation information. The insertion of functions in a program is performed by the assignment of new blocks to the corresponding identifiers. The issue is that the same function in different program will be represented by different identifiers. We solve this issue using Coq *sections*. A section is defined for each function with its associated lemmas. Its variables are defined abstractly by just giving their types. Integrating such a function into the `CompCert` code of an instruction consists in importing the file containing the corresponding section, instantiating additionally assumed variables with appropriate values, then performing memory allocation.

Proofs on library functions are performed in the same way as for instructions, see Subsection 6.4.1.

6.4.4 Proofs on tricky operations on words

We also have lemmas for checking that different ways of computing a function actually provide the same result. An example is the function for getting the bit at a

given position in a word represented by a binary integer. The equality to be proved is, after simplification:

`and (shru` x `(repr (Z_of_nat` n`))) (repr 1) =` x `[`n`].`

It means that the definition of `get_bit` used in `Simlight` can be (efficiently) computed by a combination of binary operations `and` and `shru` (logical shift right) on the integer x and the bit number n. On the right hand side, the formal specification uses a bit mask on the object integer x to get the n^{th} bit. The comparison is quite complicated due to the range of the result in type integer. We have to take a restriction into account, saying that n should be greater than 0 and less than the word size, and to add specific lemmas on other arithmetic definitions.

6.5 Tactics

Coq provides the Ltac language to allow the user to define her/his own tactics. LTac expressions can be used in the proof script of a given theorem, or in a top-level Ltac definition. The most useful construct of Ltac is the pattern matching on a proof goal, which analyzes the current goal and binds names to useful informations. This is used for our inversion `hc_inversion` described in Section 7.1. We also defined tactics dedicated to our specific needs, representing systematic reasoning schemes on the `CompCert` C semantics. Most of them deal with the C memory model and with operational semantics rules.

6.5.1 Load/store operations

Many reasoning steps are about the effect of a load/store operation on memory. Such operations are always constrained by low and high bounds of the memory blocks. In order to know whether the memory block we focus on remains the same or has been changed to a new contents, we have to determine the range of blocks targetted by operations on memory. We also need to check that a given block do not overlap with other blocks. The position of every variable and function is given during allocation. In order to find the value of blocks, we then have to analyze the appropriate allocation hypothesis, providing information on how the environment is initialized. This is performed using a series of inversions because the allocation operation is inductively defined. The

number of inversion steps is equal to the number of variables in the function. This yields the same number of new hypotheses indicating the position where each variable is allocated. The definition of the initialization of a function ensures that the blocks allocated are pairwise different from each other, and that the pointer to the next block always points to a block which has a greater position number. After a number of reasoning steps on the "less than" relation between block position numbers, we apply a suitable lemma provided by `CompCert`, `load_store_other` or `load_store_same`, to determine whether the memory state changes after a load/store operation.

6.5.2 Outcome of a statement

The execution of a statement produces an "outcome", indicating how the execution terminates: either normally or prematurely through the execution of a `[break]`, `[continue]` `[return]` statement. `Sdo` is a very common statement in `CompCert` C programs. It can be used as a wrapper of a single statement. Executing a `Sdo` statement always returns `Normal` whatever the contents is. And similarly for statement `Sskip`: it is the same as a `Sdo` with no contents. In order to manage such situations, we provide a tactic based on the inversion of the semantics of statements.

6.5.3 Function calls

We also have a tactic dedicated to function calls. It is used in all instructions, since every instruction has one or more internal function calls. This tactic aims at finding the block containing the body of the called function. Indeed, the local environment does not contain functions but only their name.

To find the function, we have to go through the global environment. The global environment is also defined using the `PTree` data structure, which maps a reference to the corresponding place in memory, or a function pointer to a function definition, or a variable pointer to the associated contents.

By analyzing the hypothesis for evaluating the function identifier we aim at, we get a hypothesis $G \vdash \texttt{find\_symbol}\ id = \lfloor b \rfloor$, saying that the global environment G contains a block b for the symbol id. Next, we invert the appropriate `eval_funcall` hypothesis: according to rule (2.14) recalled in Subsection 2.3.2, we get an hypothesis $G \vdash \texttt{find\_funct\_ptr}\ b = \lfloor f \rfloor$, saying that in the same global environment G, using the block b, then we are able to find the function pointer. Then we use the `set` and `get`

operations to explore the global environment, until we find the matching block. These proof steps are automatically performed in LTac using pattern matching on goals.

6.6 Dealing with version changes of CompCert

During the development of our correctness proofs, three versions of CompCert were released, bringing new features and better performances. The change of version from CompCert-1.8.1 to CompCert-1.9 did not cause much trouble on SimSoC-Cert. We discuss here the impact of the next two releases on our project.

6.6.1 Changes from CompCert-1.9 to 1.10

An important fact on version 1.9 is that it turned the CompCert C reduction semantics into a reference interpreter. Handling of annotation statements has been improved to separate where has one integer argument and where has arbitrarily many arguments. And efforts have been done for handling external function and compiler built-ins. The built-in external function for memory operation "copy" is now fully specified as well as other changes which we do not care. We only care about the semantics and part of the low level definitions. So, SimSoC-Cert only needs to be changed for some small point as the semantic cast is no longer a inductive type but a pattern-matching function. The way to apply such cast definition needs to follow the version. At that time, we have not begun the correctness proof involving the external functions. Then the changes due to this part can be ignored. But the version change of CompCert-1.9 to CompCert-1.10 brought backward incompatibility to SimSoC-Cert. Not only because many things have been changed in the newer version 1.10, but also our SimSoC-Cert project becomes richer and more stuff depends on CompCert, especially in the correctness proof scripts.

Next, we introduce the main changes between the two versions and explain the impact to our project.

6.6.1.1 Volatile types

CompCert C now natively supports volatile types. Its semantics fully specifies the meaning of accesses to volatile memory, and the translation of volatile accesses to built-in function invocations is proved correct.

In order to prepare future evolutions of `CompCert`, most constructors of the Coq type `type` for `CompCert` C types expect a record called `attr` (for attributes), which is introduced in `CompCert`-1.10. Volatility of memory is specified by a Boolean field in this record. Our generator had to be then changed to take this field into account. Since the introduction of volatile memory access, the way to compute the value of a given data is changed.

Introducing the volatile type also changed the definition of the projection from the concrete to the abstract representation of the ARMv6 processor, because of the use of the `load` operation in this projection.

`Simlight` currently includes no volatile variable. Then we can directly use the normal load without considering the volatile attribute. But our correctness proofs are modified, because the semantics of load/store is no longer given by a functional definition but by an inductive type, which is used to express additional concerns about the volatility of load/store. The main impact is on proofs related to assignment expressions.

6.6.1.2 Booleans

From this version, `CompCert` C provides Booleans. This could be used in `SimSoC`, where Boolean values are represented as unsigned 8-bit ints. However C Booleans are currently not considered.

6.6.2 Changes from `CompCert`-1.10 to 1.11

6.6.2.1 Memory model

The most important change here is the memory model: a more precise model of memory and permissions is defined in `CompCert`-1.11, reusing the existing module `ZMap` (a mapping from `block` to `memval` indexed by `offset`) for memory state definition, instead of using a function of type $block \rightarrow Z \rightarrow memval$. The main operations on `ZMap` are `set` and `get`. Note that `get` always returns a data: if there is no data associated with the an index given as input, a default value is returned. This default value is set when the map is initialized. For memory, the default value to be returned is "undefined". Thanks to the use of `ZMap` for memory state type, memory bounds can be dismissed.

6.6.2.2 Permission guard

Another major change is the addition of a maximal permission guard for a block, other than the one which indicates the current permission guard of a block. The maximal permission which must be stronger than the current permission, and can decrease only by freeing a block, dropping a permission of a block or performing an external function call. The corresponding field in the memory structure describing permission is then optimized.

In our development, the statement of properties of library functions mentionning the equivalence of two memory states of type `mem` needed to be changed to fit the new structure of `mem`.

Chapter 7

Designing our own inversion

In correctness proofs of ARM instructions, which involve the large-size inductively defined relation coming from `CompCert` C semantics, many steps require inverting a hypothesis to perform a case analysis and extract all useful constraints from the hypothesis. The Coq built-in tactic `inversion` is usually considered to be the right choice in such situations. But using it made us suffering from severe controllability, maintenance and efficiency issues. To circumvent these issues, we propose an inversion technique based on the combination of an antidiagonal argument and the impredicative encoding of inductive data-structures, which we are going to introduce in this chapter. Part of the material presented of this chapter has been published in (46).

Résumé

Dans les preuves de correction des instructions ARM, qui reposent sur des relations définies inductivement de grande taille, issues de la sémantique de `CompCert` C, de nombreuses étapes consistent à inverser une hypothèse pour effectuer une analyse de cas et extraire toutes les contraintes utiles contenues dans cette hypothèse. La tactique Coq standard `inversion` est généralement considérée comme étant le bon choix dans de telles situations. Cependant son utilisation nous a posé de graves problèmes de contrôlabilité, de maintenance et d'efficacité. Pour les résoudre, nous proposons une technique d'inversion basée sur la combinaison d'un argument antidiagonal et d'un codage imprédicatif des structures de données inductives, qui fait l'objet de ce chapitre. Le matériel présenté ici a été partiellement publié dans (46).

7.1 Why a new inversion

7.1.1 Inversion tactic in Coq

During the development of a proof, if a hypothesis is an instance of an inductive predicate and we want to derive the consequences of this hypothesis, the general logical principle to be used is called *inversion*. To this effect, the Coq proof assistant provides a useful tactic called `inversion` (15) which is available in several variants.

An inversion is a kind of forward reasoning step that allows for users to extract all useful information contained in a hypothesis. It is a case analysis over a given hypothesis according to its specific arguments, that removes absurd cases, introduces relevant premises in the environment and performs suitable substitutions in the whole goal. The practical need for automating inversion has been identified many years ago and most proof assistants (Isabelle, Coq, Matita,...) provide an appropriate mechanism.

7.1.2 Issue from `CompCert` C semantics

`CompCert` C semantics is a quite big and complex inductive relation. Each constructor describes the memory state transformation of an expression, statement, or function. In the theorems we aim at proving, ARM instructions are represented by C functions containing a sequence of statements which can be decomposed into complicated expressions. As soon as we want to discover the relation between memory states before and after evaluating an expression, we have to invert hypotheses of operational semantics to follow the clue given by its definition. To perform such inverting we can use `inversion`. But each use of `inversion` will go one step only.

For illustration, we present here a small excerpt from an old proof script in `SimSoC-Cert` using `inversion`, which belongs to the ADC instruction. It sets the CPSR with the value of SPSR. The pseudo-code from the ARM reference manual is just `CPSR = SPSR`. The corresponding C code is a call to function `copy_StatusRegister`, which sets CPSR field by field by the values from SPSR. Lemma `same_cp_SR` states that the C memory state of the simulator and the corresponding formal representation of ARM processor state evolve consistently during this assignment.

```
Lemma same_copy_SR :
  forall e m l b s t m' v em,
  proc_state_related m e (Ok tt (mk_semstate l b s)) ->
```

```
eval_expression (Genv.globalenv prog_adc) e m expr_cp_SR t m' v ->
forall l b,
proc_state_related m' e
 (Ok tt
     (mk_semstate l b (Arm6_State.set_cpsr s (Arm6_State.spsr s em)))).
 ).
```

After a couple of introductions and other administrative steps, we get the following goal, where `cp_SR` is unfolded in hypothesis H. `cp_SR` is the identifier of `CompCert` C representation, which calls to the function `copy_StatusRegister` with arguments CPSR as setting destination and SPSR as source.

```
l' : local
b' : bool
a' : expr
H : eval_expr (Genv.globalenv prog_adc) e m RV
       (Ecall (Evalof (Evar copy_StatusRegister T14) T14)
          (Econs
             (Eaddrof
                (Efield (Ederef (Evalof (Evar proc T3) T3) T6)
                   adc_compcert.cpsr T7) T8)
             (Econs
                (Ecall (Evalof (Evar spsr T15) T15)
                   (Econs (Evalof (Evar proc T3) T3) Enil) T8) Enil))
          T12) t m' a'
============================
 proc_state_related m' e st'
```

Then we have to invert `H` and similar generated hypotheses until all constructors used in its type are exhausted. Here 18 consecutive inversions are needed. Using `inv` tactic invented by `CompCert`, which performs standard `inversion`, clearing the inverted hypothesis, and rewriting of all auxiliary equations, the sequel of the script started as follows:

```
inv H. inv H4. inv H9. inv H5. inv H4. inv H5.
inv H15. inv H4. inv H5. inv H14. inv H4. inv H3.
inv H15. inv H5. inv H4. inv H5. inv H21. inv H13.
...
```

The old proof script includes a lot of code in this style, which makes the size of the code huge and hard to manage.

Another problem is the management of names. A single `inversion` will derive a dozen of variables and hypotheses according to the corresponding constructor in `CompCert` C formal semantics. With Coq built-in `inversion`, their names are automatically generated using consecutive numbers. This makes proof scripts highly dependent on such names. Such a feature is already not very good when writing the proof, because of the heavy use of inversions and the large number of new names generated each time. More importantly, the maintenance of proof scripts becomes a terribly awful task: each use of those uncontrolled names has to be revisited either when the formal definition of the `CompCert`-C semantics changes (upgrading from `CompCert` 1.8 to 1.9 for instance) or when the algorithm of Coq for name generation is changed (this happened from Coq 8.3 to 8.4). To provide an idea of the burden, in our first experience using Coq built-in `inversion`, the complete correctness proof on instruction ADC resulted in a file containing 2500 lines of proof scripts. Moreover, designing (and maintaining) the scripts was made uncomfortable by the compilation time of this file more than one minute most of the time was spend on `inversion`. Given that there are more than one hundred instructions in ARMv6 ISS, we considered it as urgent to find a replacement for Coq built-in `inversion`.

7.2 Design of `hc_inversion`

Here `hc_inversion` stands for *hand-crafted inversion*.

7.2.1 General design concept and example

Small inversion is a proof trick introduced in (45). It is able to perform the same as tactic `inversion` in some cases.

>From the idea of small inversion(45), we have built a more powerful inversion through several improvements and validated it to realistic applications. The following examples introduce our development step by step. To make it easy to understand, we choose a well known example about even defined for Peano's natural number. Its inductive definition is :

```
Inductive even_i : nat -> Prop :=
  | E0 : even_i 0
  | E2 : forall n, even_i n -> even_i (S (S n)).
```

As explained in (45), the main idea is to build the corresponding auxiliary diagonalization function.

First, the inductive predicate `even_i` is a dependent data type.

Using primitive tactics `case` or `destruct` is powerful enough to perform dependent pattern matching on an assumption of type `even_i n` when the conclusion of the current goal shares the same arguments as the hypothesis to be case analyzed. If not so, one cannot return the desired new goal with the converted arguments by using only `case` or `destruct`

Assume there are two proof terms `t0` and `t2` for constructors `E0` and `E2`. The two proof terms have different types. The type of `t0` is `P 0`, the type of `t2` is `P (S (S n))`. Therefore, the syntax of the `match` construct contains a `return` clause with the expected type of the result `P n` as an argument; moreover, there is also an `in` clause for the type of `H` which binds `n`:

```
match H in even_i n return P n with
  | E0 => t0
  | E2 e ex => t2
end
```

Assuming a hypothesis `H` of type `even_i n` and a conclusion of type `P n`, both sharing variable `n`, then applying a case analysis on `H` will build a proof term in the same form as the code above and generate two new sub-goals `P 0` and `P (S (S x))` with the additional assumption `even_i x`.

Sometimes, there is no obvious relation between the hypothesis and conclusion. For example, consider the following lemma: $even\_i\ 1 \rightarrow 3 = 4$, where the conclusion $(3 = 4)$ is not related to the argument of `even_i` (1). As mentioned before, our interactive destruct works only if the hypothesis we want to destruct and the conclusion share the same argument. In order to fix this, we have to convert the conclusion of the current goal into a function of 1. We define a diagonalization function `diag` which matches the key parameter and returns the conclusion of the current goal:

```
let diag x :=
  match x with
    | 1 => 3 = 4
    | _ => True
  end in
  match H in even_i n return diag n with
    | E0 => I
    | E2 _ _ => I
  end
```

Then a case analysis on `H` will return two sub-goals: `diag 0`, and `diag (S (S y))` ending up with a proof term for `True`.

However, the technique explained in the previous section has to be extended in order to cover more general situations.

The first improvement we have to provide is to make the diagonalization function independent from specific conclusion if we want it to be used for any possible goal. We use $\forall X : Prop$ instead of a specific conclusion to hook the current conclusion. Then the previous diagonalization function will be replaced. Then together with the previous proof term of type $\forall X, X$, it is able to apply any conclusion:

```
let diag x :=
  match x with
    | 1 => forall X : Prop, X
    | _ => True
  end in
  match H in even_i n return diag n with
    | E0 => I
    | E2 _ _ => I
  end
```

The second is to consider a positive case. Let us consider the following theorem as an example,

$$\forall n\ m, \texttt{even\_i n} \rightarrow \texttt{even\_i (n+m)} \rightarrow \texttt{even\_i m}.$$

The proof is led by induction on `even_i n`. According to the constructor of inductive type `even_i`, induction generates two sub-goals: `even_i (0 + m)` and `even_i (S (S`

(n + m)). The first is easy to solve. Then an induction hypothesis will be added to the local context: $even\_i\ (n+m) \to even\_i\ m$. If we want to continue, we need a link from S (S (n + m)) to n + m, and it is exactly the second constructor E2 of inductive type even_i. So we expect our technique could also express the premise of the focused constructor. We propose a new diag function and proof term defined as follows:

```
let diag x :=
  match x with
    | S (S y) => forall X: Prop, (even_i y -> X) -> X
    | _ => True
  end in
  match H in even_i n return diag n with
    | E2 p e => fun X k => k e
    | _ => I
  end
```

Then, applying the new technique in current hypothesis $H :\ even\_i(S(S(n\ +\ m)))$ yields a function in continuation passing style. The type parameter X identified to the conclusion even_i m; then y binds to n + m, and the goal converts to $even\_i\ (n+m) \to even\_i\ m$. That is exactly what we expected. Our inversion function can be seen as inversion lemmas, but their type is the dependent type expressed by their own diag. The difference between our diagonalization function and the Coq built-in Derive Inversion will be introduced at the end of this section.

To summarize this new diagonalization function, when there is an inductive type I(t), where t is the parameter of type T, and C_i is a constructor of I depending on parameter t_i of type T, pi is the premise in constructor C_i, $\mathcal{P}$ consists of a constructor of type T, we want to filter. Then a constructor of the inductive type I(t) containing $\mathcal{P}$ can be expressed like $C_i : \forall\ p_i,\ I\ \mathcal{P}$. And HI is the hypothesis of type I(t) we want to invert. In the general case, we have to consider if there are more than one possible constructors containing $\mathcal{P}$, like constructor C_i, C_j, etc. The inverting lemma corresponding to $A\mathcal{P}$ is:

```
let diag x :=
  match x with
    | P => forall X: Prop, (forall pi, X) ... (forall pj, X) -> X
    | _ =>  True
```

```
  end in
  match HI in I t return diag t with
    | Ci ei => fun X ki => ki ei
    ...
    | Cj ej => fun X kj => kj ej
    | _ => I
  end
```

Remark the close relationship with the impredicative encoding of data types in system F.

Next, we consider more than one parameter in an inductive type. The difference when we have more parameters is that using the previous inverting strategy, the identifiers for the same variable in premise and conclusion cannot be related. This problem was discovered when applying our inverting technique to the `SimSoC-Cert` project. Let us introduce a new example in order to explain the problem properly. Here is a toy language that accepts two operations: `tm_const` and `tm_plus`. The output type `val` is a natural number or a Boolean. The evaluation (`eval`) takes an argument of type `tm` and returns a value of type `val`. The Coq code is as follows:

```
Inductive tm : Type :=
  | tm_const : nat -> tm
  | tm_plus : tm -> tm -> tm.

Inductive val : Type :=
  | nval  : nat -> val
  | bval  : bool -> val.

Inductive eval : tm -> val -> Prop :=
  | E_Const : forall n,
      eval (tm_const n) (nval n)
  | E_Plus : forall t1 t2 n1 n2,
      eval t1 (nval n1) ->
      eval t2 (nval n2) ->
      eval (tm_plus t1 t2) (nval (plus n1 n2)).
```

In the inductive type `eval`, the constructor `E_Plus` has four variables: `t1`, `t2`, `n1`,

and n2. The premises and the conclusion share these variables. Without special care we lose the information of relationship of sharing.

Let us consider a theorem,

$$\forall v,\ \texttt{eval}(\texttt{tm\_plus}(\texttt{tm\_const}\ 1)\ (\texttt{tm\_const}\ 0))\ v\ \rightarrow\ v\ =\ \texttt{nval}\ 1.$$

The diagonalization function corresponding to the previous method is:

```
match x with
  | tm_plus tc1 tc2 =>
      forall X: Prop,
      (forall n1 n2, eval tc1 (nval n1) -> eval tc2 (nval n2) -> X) -> X
  | _ => True
end
```

But then, the fact that v should be `nval (plus n1 n2)` is not recorded. The solution is to add a parameter to X to keep this identification after evaluation. The modified diagonalization function for the constructor E_Plus is:

```
match x with
  | tm_plus tc1 tc2 =>
      forall X: tm -> Prop,
        (∀ n1 n2, eval tc1 (nval n1) ->
                  eval tc2 (nval n2) -> X (nval (plus n1 n2))) -> X v
  | _ => True
end
```

This example also introduces another problem we had not foreseen: a constructor may have more than one diagonalization function. Considering the same theorem as above, after inverting E_Plus, the current proof goal is:

```
n1 : nat
n2 : nat
e1 : eval (tm_const 0) (nval n1)
e2 : eval (tm_const 1) (nval n2)
============================
 nval (n1 + n2) = nval 1
```

We expect inverting e1 and e2 can give us the nat value of n1 and n2. Without any consideration, we defined the diagonalization function for E_Const like this,

```
match t with
  | tm_const n => forall (X: val -> Prop), X (nval n) -> v
  | _ => True
end
```

It chooses to keep the value for type `val`. Then we notice in current conclusion there is no `nval n1` or `nval n2` but `nval (n1 + n2)`. The previous diagonalization function is not able to get the value of `n1` or `n2`. The diagonalization function should focus on a variable of type `nat` instead of `val`. The pattern matching should match both input and output parameters of `eval`.

```
match t, v with
  | tm_const tc, nval n => forall (X: nat -> Prop), X tc -> n
  | _, _ => True
end
```

In summary, the diagonalization function is defined depending upon what conclusion we have. When we have a conclusion like in this example, we choose the second diagonalization function. If the conclusion contains only `nval n`, we can choose the first one.

7.2.2 Using our hand-crafted inversion in SimSoC-Cert

We use the new inversion to define a new inversion tactic `inv_[expr]` for inductive type `eval_expr` in `CompCert`. The semantics of `CompCert` C tells us how the memory state is transformed by evaluating expressions (Section 2.3.2). Like explained in the previous subsection, an auxiliary function has to be defined for each constructor of `eval_expr`.

First, we define the diagonal-based function for each constructor of *eval_expr*, following the lines given in the previous section. For example, the evaluation of a field is defined in `CompCert` by the following rule.

```
Inductive eval_expr :
  env -> mem -> kind -> expr -> trace -> mem -> expr -> Prop :=
  ...
  | eval_field : ∀ e m a t m' a' f ty,
      eval_expr e m RV a t m' a' ->
      eval_expr e m LV (Efield a f ty) t m' (Efield a' f ty)
```

We then define (observe that 2 variables and 1 hypothesis will be generated):

```
Definition inv_field g e m ex t m' ex'
  (ee:eval_expr g e m LV ex t m' ex') :=
  let diag e ex ex' m m' :=
    match ex with
      | Efield a b c =>
        ∀ (X:expr->Prop),
        (∀ t a', eval_expr g e m RV a t m' a' -> X (Efield a' b c)) -> X ex'
      | _ => True
    end in
    match ee in (eval_expr _ e m _ ex _ m' ex')
             return  diag e ex ex' m m'  with
      | eval_field _ _ _ t _ a' _ _ H1 =>   fun X k => k t a' H1
      | _ => I
    end.
```

Every instruction contains a quite complex expression. If we want to find the relation between the memory states affected by these expressions, we have to invert many times even if we use the new `hc_inversion`. These steps are repetitive, applying the right diagonal-based functions with the same pair of memory states as parameters to the focused hypothesis.

Using the `match goal` construct of LTac, we can define a high-level tactic for each inductive type, gathering all the functions defined for its constructors. For example, the inversion tactic for `eval_expr` contains:

```
Ltac inv_eval_expr m m' :=
  ...
  let t1_:=fresh "t" in
  let v1_:=fresh "v" in
  let ev_ex1 := fresh "ev_ex" in
  ...
  match goal with
  ...
    | [ee: eval_expr ?ge ?e m LV (Efield ?a ?f ?ty) ?t m' ?a' |- ?cl] =>
      apply (inv_field ee); clear ee; intros t1_ a1_ ev_ex1; intros;
      inv_eval_expr m m'
```

This tactic has two arguments `m` and `m'`, corresponding to C memory states. The first `intros` introduces the 3 generated components with names respectively prefixed by `t`, `v` and `ev_ex`. The second `intros` is related to previously reverted hypotheses, their names are correctly managed by Coq. The tactic proceeds as follows:

- it automatically finds the hypothesis we want to invert by matching the targetted memory states;
- related hypotheses are reverted;
- the right auxiliary function is called (all auxiliary functions are gathered in the tactic);
- meaningful names are given to derived variables and hypotheses;
- all other related hypotheses are updated according to the new names and new values;
- useless variables and hypotheses are cleaned up ;
- the steps above are repeated until all transitions between the two targetted memory states are discovered.

We name this tactic `inv_eval_expr`; all inversions on hypotheses of type `eval_expr` are replaced by `inv_eval_expr`. For example, 18 standard `inv` were used in the old proof script of lemma `same_copy_SR`. With the high-level tactic, the 18 `inv` can be replaced by one step: `inv_eval_expr m m'`.

Inverting a hypothesis of type `eval_expr` may introduce new hypotheses on internal memory states according to the premises in the definition of the constructor. The automatic naming scheme in our tactic provides useful clues which are helpful in the script of a proof. Sometimes, inverting a hypothesis will identify two memory states $\mathtt{m}_{i1}$ and $\mathtt{m}_{i2}$. Then $\mathtt{m}_{i1}$ is automatically replaced by $\mathtt{m}_{i2}$. Such replacements trouble the automatic process in our tactic, because the first memory state $\mathtt{m}_{i1}$ is used for finding the next hypothesis to be inverted. This issue is solved by inverting hypotheses in backward order.

Our `hc_inversion` makes it possible to have a convenient automatic naming algorithm because the arguments that need to be named are fixed and are known directly from the inductive type definition itself. It does not work with standard inversion because, other than the arguments and premise of the inductive definition itself, extra equalities may be introduced and hypotheses may be reordered in a way which is not under our control.

7.2.3 Comparing hc_inversion with Coq built-in inversions

There are three Coq built-in tactics that can achieve inverting the hypothesis of current proof goal. They are the standard `inversion`, `Derive Inversion`, and `dependent induction/destruction`. We already discussed the tactic `inversion`. The tactic `Derive Inversion` allows the user to first automatically generate an inversion principle according to an inductive type and then to apply it to inverting target. The tactics `dependent induction` and `dependent destruction` are another option for inverting inductive predicate instances and potentially doing induction at the same time. They are based on BasicElim of Conor McBride (41) and work by abstracting each argument of an inductive instance by a variable and constraining it by suitable equalities. The usual induction and destruct tactics can then be applied to the abstracted instance and after rewriting of the equalities, we get the expected goals.

Ease of use. If we compare these three options, without considering the issues on name control, `Derive Inversion` is the most inconvenient one. It finds the clues according to type definition of inverted hypothesis, without telling which one it matches and the returned premises are not introduced. `CompCert` defines `inv` as a combination of the standard `inversion` with substitution and clearing. So for a basic usage, it is not complicate to use. We think `BasicElim` is easier to use than the two other built-in tactics. New equalities hypotheses will be rewritten and existing premises of equation can be kept by a block. It handles the recursive type definition.

If we take name control issues into account, both `Derive Inversion` and `BasicElim` are hard to use. Names have to be provided for all cases given by the constructors. For example, we have to consider 16 cases for `eval_expr`. Even if we just use a wild-card in impossible cases, 15 wild-cards are still needed for them, as well as extra tactics for concluding.

The price we have to pay for gaining controllability and accurate management of names is that `hc_inversion` has to be updated with each release of `CompCert`. This requires some work. But as expected, proof scripts themselves are robust, changes occur only in the definitions related to `hc_inversion`. In our developments, after `hc_inversion` became available, proof scripts could also be improved much more easily,

Table 7.1: Time costs (in seconds)

	standard `inversion`	`Derive Inversion`	`BasicElim`	our inversion
Full example	1.628	0.976	1.428	0.312
Ecall	0.132	0.076	0.112	0.028
Evalof	0.132	0.072	0.092	0.020
Evar	0.128	0.064	0.084	0.024
Eaddrof	0.140	0.076	0.104	0.020

Table 7.2: Size of compilation results (in KBytes)

	standard `inversion`	`Derive Inversion`	`BasicElim`	our inversion
Full example	191	460	171	37

achieving a good separation of concerns between the design of proofs and technical issues on inversion.

Performance. Another clear advantage for our `hc_inversion` is efficiency. Proof terms generated by `hc_inversion` are much smaller than by the three built-in tactics, as shown on examples taken in `SimSoC-Cert`, see Tables 7.1 and 7.2). The comparison is performed on a lemma taken from the correctness proof of instruction ADC. This lemma discusses how the memory state changes during the evaluation of expressions including `Econdition`, `Ebinop`, `Evalof`, `Eval`, and `Evar`. We compare the time used for performing each inversion in Table 7.1, and the size of output object files (.vo) in Table 7.2.

We see that `hc_inversion` consumes 4 to 5 times less space than `inversion` and `BasicElim`, and 10 times less than `Derive Inversion_clear`. Consistently, and more importantly for the user who heavily uses inversions, `hc_inversion` reacts much faster (3 to 6 times). Note that, in our experiments, `Derive Inversion` has a better response time among the three built-in inversion tactics, but it generates the biggest .vo files.

Inversions out of reach of built-in tactics. Let us now consider a predicate defined on a dependent type. We take intervals $[1...n]$, formalized as t in the standard library `Fin`, and then we restrict them to have an odd length.

```
Inductive t : nat -> Set :=
```

```
  | F1 : forall n, t (S n)
  | FS : forall n, t n -> t (S n).

Inductive odd : forall n : nat, t n -> Prop :=
  | odd_1 : forall n, odd (S n) F1
  | odd_SS : forall n i, odd n i -> odd _ (FS (FS i)).
```

Finding the premises for the second constructor is a function similar to the one provided for $E2$ above:

```
Definition premises_odd_SS n i: t n (of: odd n i) :=
  let diag n i :=
    match i with
      | FS _ (FS _ y)  => forall (X: Prop), (odd _ y -> X) -> X
      | _ => True
    end in
  match of in odd n i return diag n i with
      | odd_SS n i o => fun X k => k o
      | _ => I
  end.
```

In particular we can easily prove:

```
Lemma odd_SS_inv: forall n i, odd _ (FS (FS i)) -> odd n i.
Proof.
  intros n i o. apply (premises_odd_SS o). trivial.
Qed.
```

Standard `inversion` happens to fail here. Note that BasicElim may work (we actually could not succeed) but would need an additional axiom related to John Major equality.

Chapter 8

Tests generator for the decoder

Currently we do not have a correctness proof of our ARM instruction decoder. Instead, we have built a decoder tests generator, which can help to check the coverage and correctness of the generated C decoder.

Résumé

Actuellement, nous n'avaons pas développé de preuve de correction pour notre décodeur d'instructions ARM intégré à `Simlight`. À la place, nous avons construit un générateur de tests pour ce décodeur, permettant de tester sa couverture et de vérifier qu'il produit des résultats corrects.

In order to validate the generated C decoder, we have built an automatic test generator that generates all possible instructions excluding undefined and unpredictable ones. We generate two kind of files. The first file contains the test instructions binary

(a) binary encoding of the `ADC` *instruction*

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 0
cond	0 0	I	0 1 0 1	S	Rn	Rd	shifter_operand

(b) binary encoding of the "logical shift left by immediate" operand

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 7	6 ... 4	3 ... 0
cond	0 0	0	opcode	S	Rn	Rd	shift_imm	0 0 0	Rm

(a+b) resulting binary encoding of the flattened instruction

31 ... 28	27 26	25	24 ... 21	20	19 ... 16	15 ... 12	11 7	6 ... 4	3 ... 0
cond	0 0	0	0 1 0 1	S	Rn	Rd	shift_imm	0 0 0	Rm

encoding in the ELF format. The second file contains the same instructions in the same order in assembly code.

The decoder has been included in a program that generates, for each instruction from the binary file, the assembly language of that decoded instruction. The second generated file can then be compared with that decoding result: the two files should be identical.

The parameter values are chosen with respect to the validity constraints to ensure that the instruction is defined and predictable. For example, the parameters of the `ADC` instruction (see Fig. 5.4) are `Rd`, `Rn`, and `shift_imm`. Binary instructions are produced with different combinations of values for them. From reading the `Syntax` and `Usage` part of each instruction, we know there are several validity constraints for some instructions. Some validity constraints are dealt with during the parameter generation. For example, register `Rn` in instruction `LDRBT` cannot be `PC` (`R15`). Hence the test chooses directly values between 0 and 14 to be assigned to `Rn`. Some other validity constraints involve two or more parameters at the same time. Continuing the example of `LDRBT`, another constraint states that `Rd` and `Rn` must be different: the generator must produce two different values and assign them to `Rd` and `Rn`. Similarly, we generate the corresponding assembly code. Under each encoding table in the reference manual, the `Syntax` part explains the syntax of the instruction, the instruction identifier, and the same parameters as in the encoding table. The contents of the generated files are shown in Figure 8.1. The left column is a group of binary test in hexadecimal format, which are legal instantiation of `ADC` instruction. The right column is their corresponding assembler code according to the syntax:

```
ADC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

They represent one group of `ADC` with under different combination of condition of execution and the value of the S bit.

We use the generated binary instruction as input for our decoder. It outputs the result in assembly code. Then using the Unix command `diff`, we can compare the decoder results and the assembly tests. Several minor issues have been detected and fixed in this way.

binary tests	asm tests
52a063ac	ADCLE R6, R0, #0xb0000002
80ab3000	ADCHI R3, R11, R0
80b0dd8f	ADCHIS SP, R0, PC, LSL #27
b0b30618	ADCLTS R0, R3, R8, LSL R6
80bb0da0	ADCHIS R0, R11, R0, LSR #27
c0bedd31	ADCGTS SP, LR, R1, LSR SP
00a157ca	ADCEQ R5, R1, R10, ASR #15
80b05251	ADCHIS R5, R0, R1, ASR R2
c0ad3268	ADCGT R3, SP, R8, ROR #4
00b55574	ADCEQS R5, R5, R4, ROR R5
a0ad806e	ADCGE R8, SP, LR, RRX

Table 8.1: Generated tests for C decoder

Chapter 9

Discussion and conclusion

We developed the certification of a part of an ARM instruction set simulator called `Simlight`, using the operational semantics of the C language provided by the `CompCert` project. Correctness proofs were performed under the interactive proof assistant Coq. A large part of the Coq specification and of the model of the simulator were automatically produced from the pseudo-code available in the ARM reference manual. A Coq proof technique for performing *inversions* was introduced in order to solve cumbersome proof steps in our work in a better way than Coq built-in tactics. Moreover, the size of proof terms generated by our our `hc_inversion` is much lower than with built-in Coq `inversion`, making Coq type checking and compilation more efficient. Additionally, we have built a test generator for the ARM instruction decoder, which generates massive tests covering all ARM instructions.

The following sections contains an assessment on the usage of operational semantics in proving the correctness of `Simlight` and the feasibility of using this new approach for proving general C programs, the overall development size of SimSoC-Cert and the TCB. We conclude with prospects of future work.

9.1 Using operational semantics for proving C programs

The certification technique we applied for `Simlight` is based on the C operational semantics provided by `CompCert`. The Coq formal representation of the C programs of each ARM instruction can be obtained from the instruction pseudo-code intermediate representation AST in two ways: either by translating it to `CompCert` C AST, or by

translating it to a textual C program, then parsing it to `CompCert` C AST using the `CompCert` C parser. In our experiments, no difference could be observed between the two approaches – no information was lost using `CompCert` C parser. `CompCert` C supports a C subset which is rich enough to describe the operations of ARM instructions.

Correctness proofs were performed using the Coq proof assistant. In this approach to the certification of C programs, the Coq proof steps in Coq are not simple. However, we were actually able to consider C programs having a large size and complex specification, using the full expressive power of Coq. Our work assesses the feasibility of using operational semantics for certifying C programs.

Proof steps related to the `CompCert` C semantics can be simplified a lot by defining Coq tactics with Ltac (the tactics language). Our initial first proof script for ADC instruction contained thousands of lines of code. Then, we identified repetitive sequences and started to define our own proof tactics in the Ltac language, resulting into much shorter proof scripts. The second version for ADC correctness proof was approximately three times smaller than the first one. In the design of these tactics, we did not seek for generality. However, since ARM instructions within the same category often have very similar statements and expressions, our tactics can actually be reused.

In Section 6.5, we have introduced more general tactics implemented in SimSoC-Cert, like finding functions in the C memory model, reusing load/store operations, etc. Those tactics are not specific to `Simlight`, they only deal with `CompCert` C semantics and memory operations. The same holds for our inversion technique: it was implemented for the needs of SimSoC-Cert as a tactic `hc_inversion` dedicated to the inductive relations defined in `CompCert` (see Section 7.2.2). However, these tactics can be reused in other projects using the same approach to the correctness proof of `CompCert` C programs, e.g. the CCCBIP project which recently started in our group and aims at building a certifying compiler from a high-level component-based language dedicated to embedded systems (BIP), with `CompCert` C as its target.

9.2 Hand-crafted inversion

Our hand-crafted inversion presented in Chapter 7 was experimented on large proofs relying on big inductive relations independently defined in the `CompCert` project. It played a key role for the success of this approach to correctness proofs of C programs,

and the extra flexibility provided by `hc_inversion` inversions could be exploited to produce smaller, more robust and manageable proofs.

It is not yet a fully automatic tactic, like the original `inversion`. We think that automation could be realized by interacting with the internals of Coq. This would be done for efficiency concerns and would not harm in the cases where the proof can be automatically completed, or is followed by tactics which do not refer to names produced by inversion.

But in a project with a big size specification like `SimSoC-Cert`, where proofs require fine tuning, interactions between the human and the proof assistant cannot be avoided. In general, in such situations, statements involve arbitrarily complex definitions, so we cannot make the assumption that decision procedures can be used. The issue is then to provide appropriate mechanisms, so that writing proof scripts and interacting with the proof assistant is made easy. We think that our hand-crafted inversion technique is a good tool in this respect: it is flexible enough for the user, practical situations can be managed with a full control on the script and valuable improvements of the script are easier to design.

Let us mention another possible application of the technique. Inversion is sometimes needed to write a function whose properties will be established later (as opposed to providing a monolithic and exhaustive Hoare-style specification and along with a VC generator such as Program). In this context, simply using the proof engine and the inversion tactic tends to generate unmanageably large terms. We can expect our technique to be very helpful in such situations.

9.3 Development size

Table 9.1 shows the size of our development. The size of the generator is almost the same as the total number of lines of the generated part for ARMv6. But note that this is the version redesigned by F. Tuong in order to be more general, so that it could be reused with other specific processors. Currently, besides ARM, it is applied to the SH4 reference manual where, instead of a specific pseudo-code, instructions are described using a C syntax.

One can note also that the generated code for the ISS takes 50 % of the Coq formal model, and almost 70 % of the C simulator. Although the gain may be considered as

Original ARM ref man (txt)	49655
ARM Parsing to an OCaml AST	1068
Generator (Simgen) for ARM	10675
Generator specifications for SH4	737
General C libraries on ARM	1852
General Coq libraries on ARM	1569
Generated C code for `Simlight` ARM operations	6681
Generated Coq code for ARM operations	2068
Generated Coq code for ARM decoding	592
Projection	857
Proof script for ADC (2011)	3171
Proof script for ADC (2012)	1204
Definition of `hc_inversion`	551
Definition of other user-defined tactics	185
Proof script for auxiliary functions	856
Proof script for BL (2012)	437
Proof script for LDRB (2012)	170
Proof script for MRS (2012)	322

Table 9.1: Sizes (in number of lines)

not that large, we think that it was worth taking this approach, given the repetitive nature of instructions.

About the proof efforts, the first experiment on the correctness of ADC costed one month. The number of Coq lines for the proof script is quite large (about 3200 for the first version), especially if we compare with the 11 lines of the corresponding pseudo-code in the reference manual. At this stage, we did not develop user-defined tactics. Now, using `hc_inversion` and other user-defined tactics, not only maintenability is much improved, but the development time for a proof is much lower. Less than one week is needed for an instruction as complicated as ADC. Until now, 11 instructions were proved correct, one from each instruction category. They are given in Table 9.2.

Category	Instruction name
branch	BL
data processing	ADC
multiply	MUL
parallel arithmetic addition and subtraction	QADD16
extended instruction	UXTAB16
miscellaneous arithmetic	CLZ
status register access	MRS
load and store	LDR
load and store multiple	LDM
semaphore	SWP

Table 9.2: ARM instructions having a correctness proof

9.4 Trusted Code Base

Our proofs depend on several tools developed elsewhere: the Coq proof assistant, the OCaml compiler and the `CompCert` C certified compiler. The TCB of these external tools have to be considered independently. Regarding Coq, the TCB is essentially its kernel.

Next, the TCB includes the formal version of the ARM reference manual on which proofs are carried on: hand-written and automatically produced Coq definitions, as described in Figure 5.1. Alternatively, automatically produced Coq definitions could be replaced by the textual reference manual (patched by our bug fixes) and Coq code generators. The TCB also includes the Coq projections from the `CompCert` C AST representation of `Simlight` code to our abstract Coq model.

9.5 Future work

The next step would be to extend the work done on `ADC` and other operations given in Table 9.2 to the full ISS. We are confident that the corresponding work on the remaining ARM instructions can then be done much faster. In particular, a number of lemmas on 14 library functions are already available. 71 library functions remain to be done.

The hand written library functions in `CompCert` C ASTs are obtained using the `CompCert` C parser. Currently, they are merged with instructions by hand, and identifiers used in these functions are added manually, in order to solve a technical issue stated on page 85. It would be better to build a "hook" which automatically finds the called functions in the parsed ASTs and generates unused block numbers for the corresponding identifiers.

We also attempted to write a Coq (functional) version of the decoder, but strong improvements are required to make it usable. The current version is based on a huge pattern-matching, which considers the 32 bits of a binary instruction in a carefully designed order. We started to design a better version of this decoder, considering the semantics of bit fields. Then, proofs for the decoder could be considered as well – automatic extraction tools from the ARM reference manual are already available. Finally, the simulation loop (basically, repeat decoding and running operations) can be be proven.

In another direction, our methodology can be reused on other processors, such as SH4.

In the future, `Simlight` 2 could be considered as well. `Simlight` 2 has adopted several optimization methods for a higher simulation speed. The most important difference is the "flattening" method applied to the instruction set (see Subsection 5.4). Some instructions are merged with their addressing mode, and the `Simlight` 2 decoder decodes the instruction and its addressing mode at the same time. Then the C definition is simpler than in `Simlight` with less function calls. We expect the proof for this `Simlight` 2 decoder to be less difficult than `Simlight`. Instruction operations in `Simlight` 2 are essentially the same as for `Simlight`. The main optimization used in `Simlight` 2 is to specialize some of the parameters according to actually used values. Therefore, one ARM instruction operation is implemented by several functions in `Simlight` 2, instead of just one function in `Simlight`; but the code of these functions is essentially the same, so there is good hope that existing correctness lemmas for `Simlight` could be restated and generalized in such a way that instances of them would just be the expected correctness lemmas for the corresponding functions in `Simlight` 2.

Our group recently started another project aiming at the implementation of certified software written in BIP, a high-level component-based language dedicated to embedded

systems, with `CompCert` C as an intermediate target. We expect the work presented is this thesis to be reused there. More generally, our implementation `hc_inversion` dedicated to `CompCert` can be re-used in any application of `CompCert` operational semantics for proving C programs. However, it has to be updated accordingly to the new releases of `CompCert`.

Version française

Introduction

Certification de SimSoC

Cette thèse expose nos travaux de certification d'une partie d'un programme C/C++ nommé `SimSoC` (Simulation of System on Chip), qui simule le comportement d'architectures basées sur des processeurs tels que ARM, PowerPC, MIPS ou SH4.

Un simulateur de *System on Chip* peut être utilisé pour développer le logiciel d'un système embarqué spécifique, afin de raccourcir les phases des développement et de test, en particulier quand la vitesse de simulation est réaliste (environ 100 millions d'instructions par seconde par cœur dans le cas de `SimSoC`). Les réductions de temps et de coût de développement obtenues se traduisent par des cycles de conception interactifs et rapides, en évitant la lourdeur d'un système de développement matériel.

Un problème critique se pose alors : le simulateur simule-t-il effectivement le matériel réel ? Pour apporter des éléments de réponse positifs à cette question, notre travail vise à prouver la correction d'une partie significative de `SimSoC`, de sorte à augmenter la confiance de l'utilisateur en ce simulateur notamment pour des systèmes critiques.

`SimSoC` est un logiciel complexe, comprenant environ 60 000 de C++, intégrant des parties écrites en SystemC et des optimisations non triviales pour atteindre une grande vitesse de simulation. La partie de `SimSoC` dédiée au processeur ARM, l'un des plus répandus dans le domaine des SoC, transcrit les informations contenues dans un manuel épais de 1138 pages. La première version du simulateur ARM de `SimSoC` a été codée à la main. Les erreurs sont inévitables à ce niveau de complexité, et certaines sont passées au travers des tests intensifs effectués sur la version précédente de `SimSoC` pour l'ARMv5, qui réussissait tout de même à simuler l'amorçage complet de Linux. Au delà des objectifs de vitesse, la précision est nécessaire. Toutes les instructions

doivent être simulées exactement comme décrit dans la spécification (en supposant que le matériel en fait de même). Dans les expériences effectuées avec `SimSoC`, des erreurs de comportement sont apparues de temps à autre mais il était très difficile d'en détecter l'origine. Des tests intensifs peuvent couvrir la plupart des instructions mais certains cas rares restent inexplorés, laissant la voie ouverte à des problèmes potentiels.

Une meilleure approche est alors nécessaire pour atteindre un degré de confiance plus élevé dans le simulateur. Nous proposons de certifier `SimSoC` par l'emploi de méthodes formelles.

Dans cette thèse, nous considérons l'un des modules de `SimSoC` : le simulateur d'architecture ARM. L'architecture ARM est l'une des plus répandues sur le marché des systèmes embarqués, en particulier les téléphones mobiles et les tablettes. Selon la compagnie ARM, 6,1 milliards processeurs ARM ont été mis sur le marché en 2010, et 95 % d'entre eux sont utilisés dans des *smart phones*.

Comme indiqué plus haut, le simulateur représente un gros volume de logiciel. Et sa spécification elle-même est assez complexe, du fait de l'architecture sophistiquée de l'ARM, qui comporte de nombreux composants. Avant de considérer l'ensemble de l'architecture ARM, nous avons décidé de concentrer nos efforts sur les parties essentielles, qui sont à la fois les plus importantes et les plus sensibles : celles qui concernent le CPU (dans la famille de processeurs ARM11). Au moment où notre travail a démarré, le module de simulation de l'ARM réalisé dans `SimSoC` correspondait à la version 5 (ARMv5). Plutôt que de certifier cette architecture bientôt dépassée, nous avons décidé d'anticiper l'évolution de `SimSoC` et de travailler sur la version suivante : ARM Version 6 (ARMv6). Pour des raisons expliquées plus bas, liées aux technologies de vérification disponibles pour C (en particulier `CompCert`), il était nécessaire de travailler sur un module écrit en C plutôt qu'en C++. Ce module, appelé `Simlight`, peut s'exécuter soit seul, soit en tant que composant intégré dans `SimSoC`.

Plus de 60 % du développement de `SimSoC` est constitué de la partie CPU (voir figure 1.1 page 9). Les parties restantes comprennent la gestion mémoire (mémoire virtuelle et pagination), la gestion des interruptions et les communications avec les périphériques. En résumé, la complexité de la cible a ainsi pu être réduite mais cela représente encore 10 000 lignes de code C à certifier. En outre la spécification reste très complexe, étant décrite dans un gros document, le manuel de référence de l'architecture ARM Version 6.

L'enjeu est donc ici de certifier un programme de cette taille par rapport à une spécification assez complexe.

Rappelons tout d'abord que la certification de programme s'appuie toujours sur un modèle formel du programme étudié. Un tel modèle formel est lui-même dérivé d'une sémantique formelle du langage de programmation utilisé. Pour des langages de programmation impératifs comme C, on considère souvent des outils basés sur une sémantique axiomatique (à la Hoare) comme `Frama-C` (13), qui intègre un ensemble d'outils d'analyse de programmes pour C. La plupart de ces outils reposent sur ACSL (ANSI/ISO C Specification Language), un langage de spécification basé sur une sémantique axiomatique de C. ACSL est assez puissant pour exprimer des propriétés directement au niveau du programme C. Des étiquettes d'état peuvent être insérées pour dénoter un point de de contrôle du programme, et peuvent être utilisées dans des fonctions logiques ou des prédicats. `Frama-C` est déjà une plate-forme assez mûre pour effectuer des analyses statiques et de la vérification déductive automatique sur des programmes C. Un avantage de `Frama-C` et des outils similaires est qu'il est supporté par des technologies de preuve automatique, qui permettent une importante économie d'efforts de la part de l'utilisateur. Des succès ont été obtenues sur des programmes complexes et astucieux, par exemple l'algorithme de Schorr-Waite qui manipule des structures chaînées.

Cependant, d'une manière générale, un haut degré d'automatisation tend à affaiblir le niveau de la certification du fait que les prouveurs automatiques sont eux-mêmes des programmes complexes et sujets à erreurs. En théorie, de tels programmes peuvent produire des certificats pouvant être contrôlés par un assistant à la preuve fiable (par exemple basé sur une architecture LCF). Mais actuellement c'est encore loin d'être le cas en pratique. Un problème supplémentaire se trouve dans la distance qui sépare la sémantique axiomatique de l'implantation effective, à moins que le générateur de VC soit lui-même certifié. Cette question a été examinée récemment, notamment dans le travail de Paolo Herms sur `WhyCert` 1.4 – et qui n'était pas disponible au moment où nous avons commencé de travailler sur la certification de `SimSoC`.

Un autre problème important est que l'automatisation n'est possible que sur des théories ou logiques offrant un pouvoir d'expression limité. Cela peut compliquer l'expression des spécifications et des propriétés attendues au bon niveau d'abstraction, notamment dans un contexte où la spécification est très complexe. Actuellement, `Frama-C`

implémente un sur-ensemble de la logique du premier ordre. Une autre limitation importante pour nous est que ACSL ne permet pas de décrire les changements de type de pointeurs (*cast*). En revanche, la sémantique opérationnelle définie pour `CompCert` C (voir ci-dessous) est capable de gérer les *casts* sur des pointeurs.

Le logiciel `Why` est l'un des composants les plus importants de `Frama-C`. Il effectue un calcul de plus faible pré-condition dans le calcul de Dijkstra. `Why` constitue la base de `Jessie`, un plugin de `Frama-C` qui compile du code C annoté en ACSL vers le langage intermédiaire de `Jessie`. Le résultat est donné en entrée au générateur de conditions de vérification (VC) de `Why`, qui à son tour produit des formules destinées à des prouveurs automatiques ou interactifs comme Coq.

La version 3 de `Why` correspond à à changement de conception en profondeur de `Why`. Il n'y a pas encore de frontal pour le langage C. Il s'agit d'une bibliothèque standard de théories logiques (arithmétique entière et réelle, opérations booléennes, ensembles, etc.) ou portant sur des structures de données de programmation (tableaux, files, tables à adressage dispersé, etc.). La transmission de code C annoté en ACSL vers le générateur de VC de `Why` 3 s'effectue par `Jessie` via un code intermédiaire en `WhyML`, un langage de spécification pour la spécification et la programmation impérative. Dans la nouvelle architecture, le langage de spécification est enrichi de façon à supporter davantage de prouveurs automatiques. En outre, une interface formelle est définie pour faciliter l'accès à de nouveaux prouveurs externes. Ainsi, choisir `Why` ou `Why` 3 dans notre cas impliquerait une dépendance vis-à-vis de la chaîne de transformation constituée de `Jessie` et `Why`, pour aller de code C annoté en ACSL jusqu'aux conditions de vérification pour Coq.

Dans le cas de `SimSoC`, nous avons besoin de prendre en compte simultanément une très grosse spécification et des astuces disponibles en C, comme le forçage de type, qui sont utilisées dans des fonctions délicates liées à la gestion de la mémoire. En d'autres termes, nous avons besoin d'un cadre de travail qui d'une part soit assez riche en mécanismes d'abstraction pour nous permettre de gérer une telle spécification, et d'autre part comporte une définition précise de suffisamment de mécanismes du langage C. Pour les raisons exposées ci-dessus, il n'était pas clair que `Frama-C` puisse satisfaire ces exigences, même avec Coq en sortie. Le calcul automatique de plus faibles pré-conditions ou de domaines de variation est peu approprié à notre cas. Nous avons besoin de vérifier des propriétés plus spécifiques relatives à la version formelle de l'architecture

ARMv6. Cette spécification est assez complexe, par exemple en ce qui concerne le principal type de données défini pour exprimer l'état du processeur.

Nous avons donc préféré essayer une approche moins classique mais plus directe, basée sur la sémantique opérationnelle de C : cela était heureusement rendu possible en théorie depuis la formalisation en Coq d'une telle sémantique au sein du projet `CompCert`. À notre connaissance, il s'agit de la première expérience de preuve de correction de programmes C à cette échelle basée sur la sémantique opérationnelle.

SimSoC

Dans cette section, nous introduisons notre cible de certification : `SimSoC`, un simulateur de *System-on-Chip* (SoC) capable de simuler divers processeurs à une vitesse réaliste. En tant que simulateur de *System-on-Chip*, les objets simulés sont des processeurs de systèmes embarqués utilisés dans des équipements modernes d'électronique grand public ou de systèmes industriels (par exemple ARM, PowerPC, MIPS). Il entre dans la catégorie des *simulateurs de système complets* car il peut simuler la plate-forme matérielle complète et exécuter le logiciel embarqué "tel quel", y compris le système d'exploitation. Ce genre de simulateur joue un rôle important dans le développement des systèmes embarqués, car le logiciel embarqué peut être testé et développé sur le simulateur. Si l'on veut que le logiciel et le matériel soient prêts à aller sur le marché en même temps, le logiciel doit parfois être développé avant que le matériel soit disponible. Un modèle exécutable du SoC est alors nécessaire. Un simulateur procure d'autres avantages, permettant de combiner la simulation avec des méthodes formelles comme le model-checking ou l'analyse de traces pour découvrir des anomalies matérielles ou logicielles.

Notre simulateur, `SimSoC`, travaille au bas niveau du système. Il prend du code binaire réel en entrée ainsi qu'un modèle de simulation de la carte complète : processeur, unités mémoire, bus, contrôleur réseau, etc. Il peut émuler le comportement de l'exécution des instructions, des exceptions et des interruptions des périphériques.

En dehors du développement de logiciel, le simulateur peut aussi être utilisé pour la conception de matériel. En présence de composants supplémentaires fournis par une tierce partie, les développeurs peuvent tester modulairement ces derniers dans l'environnement complet de simulation.

`SimSoC` est développé en SystemC, une bibliothèque C++, et utilise TLM (*transaction level modelling*) pour modéliser les communications entre les modèles de simulation. Afin de simuler les processeurs à une vitesse raisonnable, la simulation du jeu d'instructions utilise une technique appelée traduction dynamique (*dynamic translation*), qui traduit l'entrée binaire en une représentation intermédiaire elle-même compilée en code de la machine hôte. `SimSoC` étant un environnement assez gros et complexe qui influence de développement à la fois de logiciel et de matériel, il est important de comprendre quelles sont les parties les plus significatives afin de pouvoir déterminer la cible de la certification.

Simulation du jeu d'instructions

Un simulateur de systèmes complets doit inclure un simulateur de jeu d'instructions, qui lit les instructions du programme et émule le comportement du processeur cible. Pour illustrer notre cible de certification, nous détaillons ici les techniques de réalisation d'un simulateur de jeu d'instructions. Il y a trois sortes de techniques implantées pour la simulation d'instructions dans `SimSoC`, correspondant à différents compromis entre précision et efficacité. Ce sont : la *simulation par interprétation*, la *traduction dynamique sans spécialisation* et la *traduction dynamique avec spécialisation*. La *simulation par interprétation* est la méthode classique, qui comprend trois étapes : récupération de l'instruction en mémoire, décodage et exécution. Bien que cette technique soit handicapée par le décodage répété des mêmes instructions, elle est simple à réaliser et fiable. Elle sert également d'étalon de performance pour les autres techniques. La seconde et la troisième technique sont basées sur la traduction dynamique, qui utilise une représentation intermédiaire pour le résultat du décodage. Les représentations intermédiaires des instructions décodées sont stockées dans un cache et réutilisées quand les mêmes instructions doivent être ré-exécutées. La dernière méthode *traduction dynamique avec spécialisation* combine traduction dynamique et *évaluation partielle*. Cette dernière est une technique bien connue en compilation optimisante. L'idée est de traduire un programme P appliqué à une donnée spécifique d en un programme spécifique plus rapide Pd. On peut utiliser l'évaluation partielle en simulation pour spécialiser une instruction en une instruction plus simple, à partir des données connues au moment du décodage. Le décodeur de `SimSoC` implémente l'évaluation partielle. Au moment du décodage, la traduction dynamique établit une correspondance entre les instructions binaires et

leurs spécialisations partiellement évaluées. Bien que la spécialisation des instructions induise un surcroît de consommation mémoire, ce dernier reste raisonnable rapporté aux tailles mémoire disponibles sur les serveurs actuels.

Les technologies utilisées dans le simulateur de jeu d'instructions de `SimSoC` sont détaillées dans (27).

Performances

Le module ARM de `SimSoC` réalisant le jeu d'instruction de l'architecture ARMv5 a été écrit à la main. Le simulateur peut simuler le circuit commercial SPEAr Plus600, un SoC réalisé par ST Microelectronics comportant un système à double cœur et plus de 40 composants supplémentaires, ainsi que le circuit AM1705 de Texas Instruments. Le simulateur peut émuler le contrôleur d'interruptions, le contrôleur de mémoire, le contrôleur de mémoire série, le contrôleur Ethernet, et tous les périphériques nécessaires pour amorcer Linux. Exécuter le noyau Linux sur le simulateur de SPEAr Plus est un moyen de tester et mettre au point le simulateur. Tout d'abord le binaire du noyau Linux compressé est lu sur la mémoire série, décompressé, puis Linux est démarré. La simulation de ce processus ne prend que quelques secondes. Le contrôleur Ethernet peut connecter à travers le protocole TCP/IP plusieurs simulateurs du même SoC s'exécutant ou non sur la même machine. Ce simulateur mature pour l'architecture ARMv5 était terminé avant le début de notre projet de certification, et deux autres simulateurs de jeu d'instruction pour PowerPC et MIPS étaient également développés.

De l'ARMv5 à l'ARMv6

Pour cette thèse nous avons décidé de considérer la version suivante (ARMv6) de l'architecture ARM, qui représente une montée en performances depuis les cœurs ARMv5. Pour l'essentiel, ARMv6 est compatible en arrière avec ARMv5. Voici les nouveautés apparues dans l'architecture ARMv6.

– Le jeu d'instructions a été étendu par de nouvelles instructions dans six domaines : des instructions média, des instructions de multiplication, des instructions de contrôle et DSP, des instructions de chargement et stockage en mémoire, des instructions indéfinies et des instructions inconditionnelles. Heureusement, toutes les instructions ARMv5 obligatoires sont également instructions ARMv6 obligatoires. Pour les utilisateurs du simulateur, un code applicatif compilé pour

l'ARMv5 s'exécute également sur l'ARMv6. Si un utilisateur souhaite bénéficier des nouvelles instructions V6, il doit recompiler son programme dans le nouvel environnement.

– Le mode *Thumb* (instructions sur 16 bits) a changé. Les instructions *Thumb* de l'ARMv5 ne sont pas portables vers *Thumb2* (ARMv6+), la compatibilité arrière n'est pas complètement assurée.

Contributions

Dans ce travail nous avons développé une preuve de correction d'une partie du simulateur `SimSoC`. Au delà de la certification d'un simulateur, il s'agit d'une nouvelle expérience en certification de programmes non triviaux écrits en C. Contrairement aux approches répandues, nous n'utilisons pas une sémantique axiomatique mais une sémantique opérationnelle, en l'occurrence celle définie dans le projet `CompCert`.

Nous définissons une représentation du jeu d'instruction ARM et de ses modes d'adressage formalisée en Coq, grâce à un générateur automatique prenant en entrée le pseudo-code des instructions issu du manuel de référence ARM. Nous générons également l'arbre syntaxique abstrait `CompCert` du code C simulant les mêmes instructions au sein de `Simlight`, une version allégée de `SimSoC`. Une version textuelle de `Simlight`, avait été auparavant développée comme un composant de `SimSoC` par C. Helmstetter (6).

À partir de ces deux représentations Coq, nous pouvons énoncer et démontrer la correction de `Simlight`, en nous appuyant sur la sémantique opérationnelle définie dans `CompCert`. Nos premiers résultats dans cette direction sont décrits dans (57). Cette méthodologie a ensuite été appliquée à au moins une instruction de chaque catégorie du jeu d'instruction de l'ARM.

Au passage, nous avons amélioré la technologie disponible en Coq pour effectuer des *inversions*, une forme de raisonnement utilisée intensivement dans ce type de situation (46).

Nos contributions supplémentaires comprennent un générateur de décodeur pour les instructions ARM, également basé sur l'analyse du manuel de référence de l'ARM, et un générateur de tests pour le décodeur d'instructions, qui peut générer des tests massifs couvrant toutes les instructions ARM.

Conclusion

Nous avons développé la certification d'une partie de `Simlight`, un simulateur du jeu d'instructions ARM, en utilisant la sémantique opérationnelle du langage C fournie par le projet `CompCert`. Les preuves de correction ont été effectuées à l'aide de l'assistant à la preuve Coq. Une grande partie de la spécification Coq et du modèle du simulateur ont été produits automatiquement à partir du pseudo-code disponible dans le manuel de référence ARM. Une technique de preuve Coq pour effectuer des *inversions* a été introduite pour résoudre des étapes de preuve administratives de manière plus satisfaisante que par les tactiques standard de Coq comme `inversion`. Les termes de preuve engendrés par notre `hc_inversion` ont par ailleurs une taille bien plus petite qu'avec `inversion`, ce qui améliore la compilation Coq et fluidifie l'interaction avec Coq lors de la mise au point des scripts. Par ailleurs, nous avons construit un générateur de tests pour le décodeur d'instructions ARM, qui génère des tests massifs couvrant toutes les instructions ARM.

Dans la suite nous tirons quelques enseignements de notre usage de la sémantique opérationnelle pour démontrer la correction de `Simlight`, quant à la faisabilité de cette approche pour la preuve de programmes C en général. Nous donnons quelques éléments chiffrés sur le développement de `SimSoC-Cert` et précisons la base de confiance de ce projet. Nous terminons par quelques prospectives.

L'approche sémantique opérationnelle en preuve de programmes C

La technique de certification employée pour `Simlight` est basée sur la sémantique opérationnelle de C fournie par `CompCert`. La représentation formelle des programmes C pour chaque instruction ARM peut être obtenue à partir de l'AST représentant le pseudo-code de deux façons : soit en le traduisant en un AST `CompCert` C, soit en le traduisant en une forme textuelle C, qui est ensuite donnée en entrée à l'analyseur syntaxique de `CompCert` C. Dans nos expériences, les deux approches donnaient des résultats équivalents. `CompCert` C supporte un sous-ensemble de C qui est suffisamment riche pour programmer la simulation des instructions de l'ARM dans `Simlight`.

Les preuves de correction ont été effectuées dans l'assistant à la preuve Coq. Dans cette approche à la certification de programmes C, les étapes de preuve Coq ne sont

pas simples. Cependant nous avons pu considérer des programmes C ayant une spécification complexe et de grande taille, utilisant le pouvoir expressif de Coq. À partir de notre travail, nous pouvons conclure que l'approche par sémantique opérationnelle est utilisable.

Les étapes de preuve liées à la sémantique de `CompCert` C peuvent être considérablement simplifiées par la définition de tactiques dans le langage Ltac de Coq. Notre premier script de preuve pour l'instruction ADC était long de plusieurs milliers de lignes. Nous avons lors identifié les séquences répétées et commencé à définir nos propres tactiques en Ltac, ce qui a permis de raccourcir considérablement les scripts de preuve. La seconde version du script pour la preuve de correction de ADC était environ trois fois plus petite que la première. Dans la conception de ces tactiques, nous n'avons pas recherché la généralité. Cependant, comme beaucoup d'instructions ARM d'une même catégorie se ressemblent, nos tactiques peuvent en fait être réutilisées.

Dans la section 6.5, nous avons également introduit des tactiques générales pour `SimSoC-Cert`, notamment pour trouver des fonctions dans le modèle mémoire de C, réutiliser des opérations de *load/store*, etc. Ces tactiques ne sont pas spécifiques à `Simlight`, elles sont seulement relatives à la sémantique que `CompCert` C et aux opérations en mémoire. Il en va de même pour notre technique d'inversion : elle a été instanciée pour les besoins de `SimSoC-Cert` comme une tactique `hc_inversion` dédiée aux relations inductives définies dans `CompCert` (voir la section 7.2.2). Cependant ces tactiques peuvent être réutilisées dans d'autres projets suivant la même approche pour démontrer la correction de programmes `CompCert` C, par exemple le projet CCCBIP qui a démarré récemment dans note groupe et vise à construire une compilateur certifiant prenant en entrée un langage de haut niveau à base de composants pour les systèmes embarqués (BIP), avec `CompCert` C comme cible intermédiaire.

Inversion sur mesure

Notre inversion "sur mesure" (*hand crafted*) présentée au chapitre 7 a été expérimentée sur des preuves de grande taille reposant sur de grosses relations inductives définies indépendamment dans le projet `CompCert`. Cela a joué un rôle crucial pour le succès de cette approche aux preuves correction de programmes C, et le gain en flexibilité obtenu par `hc_inversion` a été exploité pour produire des preuves bien plus petites, robustes et faciles à gérer.

En l'état, ce n'est pas une tactique complètement automatique comme la tactique originale `inversion`. Nous pensons qu'une telle automatisation pourrait être réalisée en interagissant avec les mécanismes internes de Coq. Cela pourrait être effectué pour des raisons d'efficacité et serait appréciable notamment dans les cas où les sous-buts engendrés pourraient être résolus automatiquement, ou sans qu'il soit besoin de référencer les noms produits par l'inversion.

Mais dans un projet comme `SimSoC-Cert`, qui comporte une spécification de grande taille et où les preuves nécessitent des mises au point fines, les interactions entre l'utilisateur et l'assistant à la preuve ne peuvent pas être évitées. En général, dans de telles situations, les énoncés mettent en œuvre des définitions arbitrairement complexes, et on ne peut faire l'hypothèse que des procédures de décision pourront tout résoudre. Le problème est alors de fournir des mécanismes appropriés, de sorte à faciliter l'écriture des scripts de preuve et l'interaction avec l'assistant de preuve. Nous pensons que notre technique d'inversion sur mesure est un bon outil de ce point de vue : elle est suffisamment souple pour l'utilisateur, les situations pratiques peuvent être gérées en contrôlant totalement les scripts et des améliorations intéressantes des scripts sont plus faciles à concevoir.

Mentionnons une autre application possible de cette technique. Des inversions sont parfois nécessaires dans l'écriture d'une fonction dont les propriétés seront établies ultérieurement (à l'opposé du style où l'on fournit une spécification à la Hoare monolithique et exhaustive et un générateur de VC comme Program). Dans ce contexte, une utilisation simple du moteur de preuve et de la tactique `inversion` a tendance à générer des termes extrêmement gros et compliqués à gérer. Notre technique devrait s'avérer très utile dans ce genre de situation.

Taille du développement

La table F.1 indique la taille de notre développement. La taille du générateur atteint presque le nombre total de lignes des parties générées pour l'ARMv6. Mais il faut noter qu'il s'agit ici de la version courante refaite par F. Tuong pour généraliser le procédé à d'autres processeurs. Actuellement, en dehors de l'ARM, cette chaîne de génération est également appliquée au manuel de référence du processeur SH4 où, à la place d'un pseudo-code spécifique, les instruction sont décrites dans la syntaxe du langage C.

Manuel de référence ARM original (txt)	49655
Analyseur ARM vers AST OCaml	1068
Générateur (Simgen) pour l'ARM	10675
Générateur de spécifications pour SH4	737
Bibliothèques générales C sur l'ARM	1852
Bibliothèques générales Coq sur l'ARM	1569
Code C généré pour les opérations ARM `Simlight`	6681
Code Coq généré pour les opérations ARM	2068
Code Coq généré pour le décodeur ARM	592
Projection	857
Script de preuve pour ADC (2011)	3171
Script de preuve pour ADC (2012)	1204
Définition de `hc_inversion`	551
Autres tactiques	185
Script de preuve pour fonctions auxiliaires	856
Script de preuve pour BL (2012)	437
Script de preuve pour LDRB (2012)	170
Script de preuve pour MRS (2012)	322

Table F.1: Tailles (en nombre de lignes)

On peut également noter que le code généré pour le simulateur de jeu d'instructions occupe 50 % du modèle formel Coq, et presque 70 % du simulateur C.

Bien que le gain en volume puisse être considéré comme relativement faible, nous pensons que cette approche est néanmoins valable étant donnée la nature répétitive des instructions.

En ce qui concerne l'effort de preuve, la première expérience a porté sur la correction de l'instruction ADC et cela nous a pris un mois. Le nombre de lignes Coq pour le script de preuve était alors assez grand, environ 3200 pour cette première version (surtout si l'on compare aux 11 lignes du pseudo-code correspondant dans le manuel de référence). À ce stade nous n'avions pas encore développé de tactiques utilisateur. Ensuite, en utilisant `hc_inversion` et nos autres tactiques spécifiques, en dehors des gains en taille et en maintenabilité, le temps de développement pour la preuve d'une instruction est bien plus faible, moins d'une semaine pour une instruction du même degré de complexité que ADC. A présent, nous avons une preuve de correction pour 11 instructions, une dans

chaque catégorie d'instructions pour l'ARM. Elles sont présentées dans la table F.2.

Catégorie	Nom de l'instruction
branchement	BL
calcul	ADC
multiplication	MUL
addition et soustraction en arithmétique parallèle	QADD16
instruction étendue	UXTAB16
arithmétique divers	CLZ
accès au registre de status	MRS
chargement et stockage	LDR
chargement et stockage multiple	LDM
sémaphore	SWP

Table F.2: Instructions ARM avec une preuve de correction

Base de confiance du code

Nos preuves dépendent de plusieurs outils développés ailleurs : l'assistant à la preuve Coq, le compilateur OCaml et le compilateur certifié `CompCert` C. La base de confiance de ces outils doit être considérée indépendamment. En ce qui concerne Coq, il s'agit essentiellement du noyau.

Ensuite, la base de confiance comprend la version formelle du manuel de référence servant à formuler les théorèmes de correction : il s'agit de définitions Coq produites manuellement et automatiquement, selon le procédé décrit en figure 5.1. Une alternative pourrait être de remplacer les définitions Coq produites automatiquement par le manuel de référence textuel (corrigé par nos soins) et la chaîne de génération de code Coq.

La base de confiance comprend enfin les projections définies en Coq entre la représentation du code de `Simlight` sous forme d'AST `CompCert` C et notre modèle Coq abstrait.

Travaux futurs

La prochaine étape serait d'étendre le travail effectué sur `ADC` et les autres opérations données dans la table F.2 au jeu d'instruction entier. Nous somme confiants dans le fait que le travail correspondant peut être effectué beaucoup plus vite. En particulier,

des lemmes portant sur 14 fonctions de la bibliothèque sont déjà disponibles. Il reste 71 fonctions semblables dans la librairie.

Les AST des fonctions de la bibliothèque sont obtenus au moyen de l'analyseur syntaxique pour `CompCert` C. Ces AST sont actuellement regroupés à la main avec les AST des instructions, pour résoudre le problème technique mentionné page 85. Il serait préférable d'introduire un mécanisme permettant de trouver automatiquement les fonctions appelées dans les AST produits par l'analyse syntaxique et de générer des numéros de blocs inutilisés pour les identificateurs correspondants.

Nous avons également tenté d'écrire une version Coq (fonctionnelle) du décodeur, mais des améliorations importantes sont nécessaires pour le rendre utilisable. La version courante est basée sur un filtrage géant, qui considère les 32 bits d'une instruction binaire dans un ordre soigneusement mis au point. Nous avons commencé à concevoir une meilleure version de ce décodeur, en considérant la sémantique des champs de bits. Cela fait, des preuves sur le décodeur pourront également être considérées. Les outils d'extraction automatique du codage des instructions décrit dans le manuel de référence sont déjà disponibles. Enfin, la correction de la boucle de simulation (essentiellement : répéter le décodage et l'exécution des opérations) pourra être prouvée.

Dans une autre direction, notre méthodologie peut être réutilisée sur d'autres processeurs, comme le SH4.

Par la suite, on pourra également considérer `Simlight` 2, qui comporte plusieurs optimisations en vue d'accélérer la simulation. La différence la plus importante est l'application d'une méthode de *flattening* (voir § 5.4) consistant à fusionner le code des modes d'adressage dans certaines instructions. Le décodeur de `Simlight` 2 décode alors simultanément une instruction et son mode d'adressage. Cela rend la définition en C plus simple que dans `Simlight` et produit moins d'appels de fonctions. Les preuves correspondantes pour le décodeur devraient donc êtr plus simples pour `Simlight` 2 que pour `Simlight`. Le corps des instructions est essentiellement le même dans `Simlight` 2 que dans `Simlight`. La principale optimisation effectuée dans `Simlight` 2 consiste à spécialiser certains paramètres aux valeurs effectives utilisées. Ainsi, l'opération d'une instruction ARM est implémentée par plusieurs fonctions dans `Simlight` 2, là où il n'y a qu'une seule fonction dans `Simlight` ; mais le code de ces fonctions est essentiellement le même, ce qui donne bon espoir à la possibilité de réutiliser les lemmes de correction existants pour `Simlight`, en les reformulant et généralisant de manière adéquate, de

sorte que par instanciation on retrouve les lemmes de correction attendus pour les fonctions correspondantes dans `Simlight` 2.

Notre équipe a récemment démarré un autre projet visant l'implantation de logiciel certifié écrit en BIP, un langage de haut niveau à base de composants dédié aux systèmes embarqués, avec `CompCert` C comme langage intermédiaire. Nous avons bon espoir que le travail présenté dans cette thèse pourra être réutilisé. Plus généralement, notre réalisation de `hc_inversion` pour `CompCert` peut être réutilisée dans toute application visant à prouver la correction de programmes C à partir de la sémantique opérationnelle définie dans `CompCert`. Cependant, cette tactique doit évoluer en fonction des nouvelles versions de `CompCert`.

Appendices

Appendix A

Example: the complete ADC instruction in Simlight

Here is the complete `CompCert` C code for simulating the ADC instruction (Add with Carry) in `Simlight`. This program contains a pretty-printed version of the `CompCert` C AST which was automatically derived from the pseudo-code for ADC given in the ARMv6 reference manual, using the generator described in Chapter 5. together with library functions, which were written by hand according to specifications in the ARMv6 reference manual.

A. EXAMPLE: THE COMPLETE ADC INSTRUCTION IN SIMLIGHT

```
struct SLv6_MMU;
struct SLv6_Processor;
struct SLv6_StatusRegister;
struct SLv6_SystemCoproc;

struct SLv6_MMU {
  unsigned int begin;
  unsigned int size;
  unsigned int end;
  unsigned char *mem;
};

struct SLv6_Processor {
  struct SLv6_MMU *mmu_ptr;
  struct SLv6_StatusRegister cpsr;
  struct SLv6_StatusRegister spsrs[5];
  struct SLv6_SystemCoproc cp15;
  unsigned int id;
  unsigned int user_regs[16];
  unsigned int fiq_regs[7];
  unsigned int irq_regs[2];
  unsigned int svc_regs[2];
  unsigned int abt_regs[2];
  unsigned int und_regs[2];
  unsigned int *pc;
  unsigned char jump;
};

struct SLv6_StatusRegister {
  unsigned char N_flag;
  unsigned char Z_flag;
  unsigned char C_flag;
  unsigned char V_flag;
  unsigned char Q_flag;
  unsigned char J_flag;
  unsigned char GE0;
  unsigned char GE1;
  unsigned char GE2;
  unsigned char GE3;
  unsigned char E_flag;
  unsigned char A_flag;
  unsigned char I_flag;
  unsigned char F_flag;
  unsigned char T_flag;
  int mode;
  unsigned int background;
};

struct SLv6_SystemCoproc {
  unsigned char ee_bit;
  unsigned char u_bit;
  unsigned char v_bit;
};

unsigned char const __stringlit_6[71] = "ERROR: simulating something
    unpredictable (../arm6/simlight/adc.c:%d)\012";
unsigned char const __stringlit_3[49] = "!(new_pc&(inst_size(proc)-1)) &&
    \042pc misaligned\042";
unsigned char const __stringlit_2[11] = "reg_id!=15";
unsigned char const __stringlit_4[14] = "pc misaligned";
```

```
unsigned char const __stringlit_5[81] = "ERROR: Current mode does not
    have a SPSR (../arm6/simlight/slv6_processor.h:%d)\012";
unsigned char const __stringlit_1[34] =
    "../arm6/simlight/slv6_processor.h";

extern unsigned char ConditionPassed(struct SLv6_StatusRegister *, int);
extern void copy_StatusRegister(struct SLv6_StatusRegister *, struct
    SLv6_StatusRegister *);
extern unsigned int *addr_of_reg_m(struct SLv6_Processor *, unsigned
    char, int);

unsigned int reg_m(struct SLv6_Processor *proc, unsigned char reg_id, int
    m)
{
  return *addr_of_reg_m(proc, reg_id, m);
}

void set_reg_m(struct SLv6_Processor *proc, unsigned char reg_id, int m,
    unsigned int data)
{
  *addr_of_reg_m(proc, reg_id, m) = data;
}

unsigned int reg(struct SLv6_Processor *proc, unsigned char reg_id)
{
  return reg_m(proc, reg_id, (*proc).cpsr.mode);
}

void set_reg(struct SLv6_Processor *proc, unsigned char reg_id, unsigned
    int data)
{
  reg_id != 15 ? (void) 0
    : __assert_fail(__stringlit_2, __stringlit_1, 58, (unsigned char *)
        0);
  set_reg_m(proc, reg_id, (*proc).cpsr.mode, data);
}

unsigned int inst_size(struct SLv6_Processor *proc)
{
  return (*proc).cpsr.T_flag ? 2 : 4;
}

void set_pc_raw(struct SLv6_Processor *proc, unsigned int new_pc)
{
  (!(new_pc & inst_size(proc) - 1) ? (__stringlit_4 ? 1 : 0) : 0) ?
      (void) 0
    : __assert_fail(__stringlit_3, __stringlit_1, 68, (unsigned char *)
        0);
  (*proc).jump = 1;
  *(*proc).pc = new_pc + 2 * inst_size(proc);
}

void set_reg_or_pc(struct SLv6_Processor *proc, unsigned char reg_id,
    unsigned int data)
{
  if (reg_id == 15) {
    set_pc_raw(proc, data);
```

```
  } else {
    set_reg(proc, reg_id, data);
  }
}

unsigned char CurrentModeHasSPSR(struct SLv6_Processor *proc)
{
  return (*proc).cpsr.mode < 5;
}

struct SLv6_StatusRegister *spsr(struct SLv6_Processor *proc)
{
  if (CurrentModeHasSPSR(proc)) {
    return &*((*proc).spsrs + (*proc).cpsr.mode);
  } else
    ERROR("Current mode does not have a SPSR");
  abort();
}

unsigned char CarryFrom_add2(unsigned int a, unsigned int b)
{
  return a + b < a;
}

unsigned char CarryFrom_add3(unsigned int a, unsigned int b, unsigned int
    c)
{
  return CarryFrom_add2(a, b) ? 1 : (CarryFrom_add2(a + b, c) ? 1 : 0);
}

unsigned char OverflowFrom_add2(unsigned int a, unsigned int b)
{
  unsigned int r;
  r = a + b;
  return ((a ^ ~b) & (a ^ r)) >> 31;
}

unsigned char OverflowFrom_add3(unsigned int a, unsigned int b, unsigned
    char c)
{
  return OverflowFrom_add2 (a, b) || OverflowFrom_add2 (a + b, c);
}

unsigned char get_bit(unsigned int x, unsigned int n)
{
  return x >> n & 1;
}
```

```
void ADC(struct SLv6_Processor *proc, unsigned char S, int cond, unsigned
    char d, unsigned char n, unsigned int shifter_operand)
{
  unsigned int old_Rn;
  unsigned int old_CPSR;
  old_Rn = reg(proc, n);
  old_CPSR = (*proc).cpsr;
  if (ConditionPassed(&(*proc).cpsr, cond)) {
    set_reg_or_pc(proc, d, old_Rn + shifter_operand + old_CPSR.C_flag);
    if (S == 1 ? (d == 15 ? 1 : 0) : 0) {
      if (CurrentModeHasSPSR(proc)) {
        copy_StatusRegister(&(*proc).cpsr, spsr(proc));
      } else
        unpredictable();
    } else {
      if (S == 1) {
        (*proc).cpsr.N_flag = get_bit(reg(proc, d), 31);
        (*proc).cpsr.Z_flag = reg(proc, d) == 0 ? 1 : 0;
        (*proc).cpsr.C_flag =
          CarryFrom_add3(old_Rn, shifter_operand, old_CPSR.C_flag);
        (*proc).cpsr.V_flag =
          OverflowFrom_add3(old_Rn, shifter_operand, old_CPSR.C_flag);
      }
    }
  }
}
```

A. EXAMPLE: THE COMPLETE ADC INSTRUCTION IN SIMLIGHT

Appendix B

Example: the proof script related to instruction ADC

Here we present Coq code on the main theorem stated for ARM instruction ADC. There are 3 memory state transitions for the concrete model. First, from `m0` to `m1`, the parameters of ADC is allocated. Second, from `m1` to `m2`, the parameters are initialized. From this memory state `m2`, we are able to build the projection to the abstract model for processor state `proc` and other parameters. Then, from `m2` to `mfin`, the statement of ADC function body is executed. The new abstract state is `S.ADC_step sbit cond d n so (mk_semstate nil true s)`. It is expected to be related to `mfin` in the concrete model.

```
Theorem related_aft_ADC: forall e m0 m1 m2 mfin vargs s out sbit cond d n so,
  alloc_variables empty_env m0 (fun_internal_ADC.(fn_params) ++
                                fun_internal_ADC.(fn_vars)) e m1 ->
  bind_parameters e m1 fun_internal_ADC.(fn_params) vargs m2 ->
  (forall m ch b ofs, Mem.valid_access m ch b ofs Readable) ->
  proc_state_related proc m2 e (Ok tt (mk_semstate nil true s)) ->
  sbit_func_related m2 e sbit ->
  cond_func_related m2 e cond ->
  d_func_related m2 e d ->
  n_func_related m2 e n ->
  so_func_related m2 e so ->
  exec_stmt (Genv.globalenv prog_adc) e m2 fun_internal_ADC.(fn_body)
```

```
    Events.E0 mfin out ->
  proc_state_related proc mfin e
    (S.ADC_step sbit cond d n so (mk_semstate nil true s)).
```

The proof script for theorem `related_aft_ADC` is too long to be present here ($\sim$ 600 loc). Instead of showing the whole script, we choose one of the lemmas used to support the proof of `related_aft_ADC`: `same_copy_SR`.

Before stating a lemma, in order to shorten the proof script and its readability, we give a name to the expression we focus on for the lemma.

The name `cp_SR` is given to the ASTs of C expression:

```
copy_StatusRegister(&proc->cpsr, spsr(proc))
```

In this expression, we have two function calls to `spsr` and `copy_StatusRegister`.

```
Definition cp_SR :=
  (Ecall
    (Evalof (Evar copy_StatusRegister T32) T32)
    (Econs
      (Eaddrof
        (Efield
          (Evalof
            (Ederef
              (Evalof (Evar proc T2) T2) T8)
            T8) cpsr T9) T25)
      (Econs
        (Ecall (Evalof (Evar spsr T33) T33)
          (Econs (Evalof (Evar proc T2) T2)
            Enil) T25) Enil)) T10).
```

The Lemma states that the evaluation results of expression `cp_SR` in the abstract model and the concrete model are equivalent.

```
Lemma same_copy_SR :
  forall e m l b s t m' v em,
    proc_state_related m e (Ok tt (mk_semstate l b s)) ->
    eval_expression (Genv.globalenv prog_adc) e m cp_SR t m' v ->
```

```
    proc_state_related m' e
      (Ok tt (mk_semstate l b
      (Arm6_State.set_cpsr s (Arm6_State.spsr s em)))).
Proof.
  intros until em. intros psrel cpsr.
  inversion cpsr. subst. rename H into ee,H0 into esrv. unfold cp_SR in ee.
  inv_eval_expr m m'.
  (* Function spsr, get spsr from the current state *)
  apply (same_spsr e l b s vf0 fd0 m vargs0 t10 m3 vres0 l b s)
    in psrel; [idtac|exact Heqff0|exact ev_funcall].
  (* Function copy_StatusRegister, copy the current spsr to cpsr*)
  apply (same_copy e l b s vf fd m3 vargs t2 m' vres l b
    (Arm6_State.set_cpsr s (Arm6_State.spsr s em))) in psrel;
  [idtac|exact Heqff|exact ev_funcall0].
  exact psrel.
Qed.
```

Bibliography

[1] Iso. International standard ISO/IEC 9899:1990, Programming language-C, 1990. 37, 42

[2] ARM. *ARM Architecture Reference Manual DDI 0100I.* ARM, 2005. 41, 63

[3] D. August and al. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *Computer Architecture Letters*, 6(2):45–48, Feb. 2007.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004. 29

[6] F. Blanqui, C. Helmstetter, V. Joloboff, J.-F. Monin, and X. Shi. Designing a CPU model: from a pseudo-formal document to fast code. In *Proceedings of the 3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, Heraklion, Greece, January 2011. 8, 22, 130

[7] F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. *Boogie*, 2011:53–64, 2011. 10

[8] R. Boyer and J. Moore. *A computational logic.* ACM monograph series. Academic Press, 1979. 15

[9] R. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM (JACM)*, 43(1):166–192, 1996. 15

[10] B. Brock, M. Kaufmann, and J. Moore. Acl2 theorems about commercial microprocessors. In *Formal Methods in Computer-Aided Design*, pages 275–293. Springer, 1996. 15

[11] P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors. *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*. Springer, 2002. 155

[12] B. Campbell. An executable semantics for compcert c. In *Certified Programs and Proofs*, pages 60–75. Springer, 2012. 25

[13] G. Canet, P. Cuoq, and B. Monate. A value analysis for c programs. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 123–124. IEEE, 2009. 9, 125

[14] A. Chlipala. Certified Programming with Dependent Types, 2012. Available at `http://adam.chlipala.net/cpdt`.

[15] Coq Development Team. *The Coq Reference Manual, Version 8.2*. INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`. 29, 96

[16] C. Cornes and D. Terrasse. Automating inversion of inductive predicates in coq. In *TYPES*, pages 85–104, 1995.

[17] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. *Frama-C User Manual, Release Boron-20100401*. CEA LIST, Software Reliability Laboratory, Saclay, France, 2010.

[18] D. Delahaye. A Tactic Language for the System Coq. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 85–95, Reunion Island (France), Nov. 2000. Springer.

[19] F. Engel, J. Nührenberg, and G. P. Fettweis. A generic tool set for application specific processor architectures. In *Proceedings of the eighth international workshop on Hardware/software codesign*, CODES '00, pages 126–130, New York, NY, USA, 2000. ACM.

[20] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification, Lecture Notes in Computer Science 4590*, 2007. 20

[21] A. C. J. Fox and M. O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP*, pages 243–258, 2010. 19

[22] F. Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005. ISBN 0-387-26232-6.

[23] S. Goel and W. Hunt. Automated code proofs on a formal model of the x86. In *VSTTE*, 2013. 17

[24] G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008. 34

[25] G. Gonthier. Engineering mathematics: the odd order theorem proof. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–2. ACM, 2013. 34

[26] C. Helmstetter and V. Joloboff. Simsoc: A systemc tlm integrated iss for full system simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 1759–1762. IEEE, 2008. 7

[27] C. Helmstetter, V. Joloboff, and H. Xiao. SimSoC: A full system simulation software for embedded systems. In IEEE, editor, *OSSC'09*, 2009. 13, 129

[28] P. Herms. *Certification of a Tool Chain for Deductive Program Verification*. PhD thesis, Université de Paris-Sud, January 2013. 20

[29] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. *Verified Software: Theories, Tools, Experiments*, pages 2–17, 2012. 20

[30] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNAI*. Springer-Verlag, 1994.

[31] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. 21

[32] U. Kuhne, S. Beyer, and C. Pichler. Generating an efficient instruction set simulator from a complete property suite. In *Rapid System Prototyping, 2009. RSP'09. IEEE/IFIP International Symposium on*, pages 109–115. IEEE, 2009. 18

[33] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[34] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[35] X. Leroy. *The CompCert C verified compiler. Documentation and user's manual*. INRIA Paris-Rocquencourt, March 2012. http://creativecommons.org/licenses/by-nc-sa/3.0/. 34, 35, 41

[36] X. Leroy, W. Appel, Andrew, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Rapport de recherche RR-7987, INRIA, June 2012.

[37] X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.*, 41(1):1–31, 2008. 38

[38] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[39] R. Leupers, J. Elste, B. Landwehr, and B. L. Generation of interpretive and compiled instruction set simulators. In *in: Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 339–342, 1999.

[40] P. S. Magnusson and al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[41] C. McBride. Inverting inductively defined relations in lego. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs*, volume 1512 of *Lecture Notes in Computer Science*, pages 236–253. Springer Berlin Heidelberg, 1998. 107

[42] C. McBride. Elimination with a Motive. In Callaghan et al. (11), pages 197–216.

[43] J. McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

[44] W. S. Mong and J. Zhu. A retargetable micro-architecture simulator. *Design Automation Conference*, 0:752, 2003.

[45] J.-F. Monin. Proof trick: Small inversions. In Y. Bertot, editor, *Second Coq Workshop*, Royaume-Uni Edinburgh, July 2010. Yves Bertot. 98, 99

[46] J.-F. Monin and X. Shi. Handcrafted Inversions Made Operational on Operational Semantics. In S. Blazy, C. Paulin, and D. Pichardie, editors, *Interactive Theorem Proving*, volume 0000 of *LNCS*, pages 000–000, Rennes, France, July 2013. Springer. 22, 95, 130

[47] G. Morrisett, G. Tan, J. Tassarotti, J. Tristan, and E. Gan. Rocksalt: Better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 395–404. ACM, 2012. 17, 18

[48] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992. 25

[49] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 22–27, New York, NY, USA, 2002. ACM.

[50] Open SystemC Initiative. *SystemC v2.2.0 Language Reference Manual (IEEE Std 1666-2005)*, 2006. `http://www.systemc.org/`.

[51] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, 2010.

[52] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. *SIGPLAN Not.*, 39:47–56, June 2004.

[53] M. Reshadi, N. Dutt, and P. Mishra. A retargetable framework for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst.*, 5(2):431–452, 2006.

[54] W. Ricciotti. *Theoretical and Implementation Aspects in the Mechanization of the Metatheory of Programming Languages*. PhD thesis, Università di Bologna, 2011.

[55] E. C. Schnarr, M. D. Hill, and J. R. Larus. Facile: a language and compiler for high-performance processor simulators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 321–331, New York, NY, USA, 2001. ACM.

[56] T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6):663–697, 2009. 48

[57] X. Shi, J.-F. Monin, F. Tuong, and F. Blanqui. First Steps Towards the Certification of an ARM Simulator Using Compcert. In J.-P. Jouannaud and Z. Shao, editors, *Certified Proofs and Programs*, volume 7086 of *LNCS*, pages 346–361, Kenting, Taiwan, December 2011. Springer. 22, 130

[58] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *DATE'99*, page 62, New York, NY, USA, 1999. ACM.

[59] J. Zhu and D. D. Gajski. An ultra-fast instruction set simulator. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):363–373, 2002.