First steps towards the certification of an ARM simulator using Compcert

Xiaomu Shi, Jean-François Monin, Frederic Tuong, Frédéric Blanqui

First Steps Towards the Certification of an ARM Simulator

Xiaomu Shi[1], Frédéric Blanqui[3], Jean-François Monin[1,2], and Frédéric Tuong[3]

[1] Université de Grenoble 1 - LIAMA
[2] CNRS - LIAMA
[3] INRIA - LIAMA

Abstract. The simulation of Systems-on-Chip (SoC) is nowadays a hot topic because, beyond providing many debugging facilities, it allows the development of dedicated software before the hardware is available. Low-consumption CPUs such as ARM play a central role in SoC. However, the effectiveness of simulation depends on the faithfulness of the simulator. To this effect, we propose here to prove significant parts of such a simulator, SimSoC. Basically, on one hand, we develop a Coq formal model of the ARM architecture while on the other hand, we consider a version of the simulator including components written in Compcert-C. Then we prove that the simulation of ARM operations, according to Compcert-C formal semantics, conforms to the expected formal model of ARM. Size issues are partly dealt with using automatic generation of significant parts of the Coq model and of SimSoC from the official textual definition of ARM. However, this is still a long-term project. We report here the current stage of our efforts and discuss in particular the use of Compcert-C in this framework.

1 Introduction

1.1 Simulation of Systems-on-Chip

Systems-on-Chip (SoC), used in devices such as smart-phones, contain both hardware and software. A part of the software is generic and can be used with any hardware systems, and thus can be developed on any computer. In contrast, developing and testing the SoC-specific code can be done only with this SoC, or with a *software executable model* of the SoC. To reduce the time-to-market, the software development must start before the hardware is ready. Even if the hardware is available, simulating the software on a model provides more debugging capabilities.

The fastest simulators use native simulation. The software of the *target* system (i.e., the SoC) is compiled with the normal compiler of the computer running the simulator, but linked with special system libraries. Examples of such simulators are the Android and iOS SDKs.

In order to develop low-level system code, one needs a simulator that can take the real binary code as input. Such a simulator requires a model of the processor and of its peripherals (such as UART, DMA, Ethernet card, etc). When

simulating a smart-phone SoC, this kind of functional simulator can be from 1 to 10 times slower than the real chip. These simulators have other uses, for example, as reference models for the hardware verification. An error in the simulator can then mislead both the software and the hardware engineers. QEMU [2] is an open-source processor emulator coming with a set of device models; it can simulate several operating systems. Other open-source simulators include UNISIM [1] (accurately-timed) and SimSoC [8], developed by some colleagues, which is loosely-timed (thus faster). Simics [12] is a commercial alternative. The usual language to develop such simulators is C++, combined with the SystemC [13] and OSCI-TLM [15] libraries.

The work reported here is related to SimSoC.

1.2 The Need for Certification

Altogether, a functional simulator is a complex piece of software. SimSoC, which is able to simulate Linux both on ARM and PowerPC architectures at a realistic speed (over over 10 Millions instructions per second), includes about 60,000 lines of C++ code. The code uses complex features of the C++ language and of the SystemC library. Moreover, achieving high simulation speeds requires complex optimizations, such as dynamic translation [2].

This complexity is problematic, because beyond speed, *accuracy* is required: all instructions have to be simulated exactly as described in the documentation. There is a strong need to strengthen the confidence that simulations results match the expected accuracy. Intensive tests are a first answer. For instance, as SimSoC is able to run a Linux kernel on top of a simulated ARM, we know that many situations are covered. However it turned out, through further experiments, that it was not sufficient: wrong behaviors coming from rare instructions were observed after several months. Here are the last bugs found and fixed by the SimSoC team while trying to boot Linux on the SPEArPlus600 SoC simulator.

- After the execution of an `LDRBT` instruction, the contents of the base register (`Rn`) was wrong. It was due to a bug in the reference manual itself; the last line of the pseudo-code has to be deleted.
- After a data abort exception, the base register write-back was not canceled.
- Additionally, a half-word access to an odd address while executing some SPEArPlus600 specific code was not properly handled.

Therefore we propose here to certify the simulator, that is, to prove, using formal methods – here: the Coq proof assistant [5, 3] – that it conforms to the expected behavior.

This is a long term goal. Before going to the most intricate features of a simulator such as SimSoC, basic components have to be considered first. We then decided to focus our efforts on a sensible and important component of the system: the CPU part of the ARMv6 architecture (used by the ARM11 processor family). This corresponds to a specific component of the SimSoC simulator, which was previously implementing the ARMv5 instruction set only. Rather than certifying

this component, it seemed to us more feasible to design a new one directly in C, in such a way that it can be executed alone, or integrated in SimSoC (by including the C code in the existing C++ code). We call this new component `simlight` [4]. Combined with a small `main` function, `simlight` can simulate ARMv6 programs as long as they do not access any peripherals (excepted the physical memory) nor coprocessors. There is no MMU (Memory Management Unit) yet. Integrating it in SimSoC just requires to replace the memory interface and to connect the interrupts (IRQ and FIQ) signals.

The present paper reports our first efforts towards the certification of `simlight`. We currently have a formal description of the ARMv6 architecture, a running version of `simlight`, and we are in the way of performing correctness proofs. The standard way for doing this is to use Hoare logics or a variant thereof. Various tools exist in this area, for example Frama-C [6]. We chose to try a more direct way, based on an operational semantics of C; more precisely the semantics of Compcert-C defined in the Compcert project [10]. One reason is that we look for a tight control on the formulation of proof obligations that we will have to face. Another advantage is that we can consider the use of the certified compiler developed in Compcert, and get a very strong guarantee on the execution of the simulator (but then, sacrificing speed to some extent[4]).

Another interesting feature of our work is that the most tedious (hence error prone) part of the formalization – the specification of instructions – is automatically derived from the reference manual. It is well known that the formal specification of such big applications is the main weak link in the whole chain. Though our generators cannot be proved correct, because the statements and languages used in the reference manual have no formal semantics, we consider this approach as much more reliable than a manual formalization. Indeed, a mistake in a generator will impact several or all operations, hence the chances that it will be detected through a visibly wrong behavior are much higher than with a manual translation, where a mistake will impact only one (eventually rarely used) operation.

Note that after we could handle the full set of ARM instructions, our colleagues of the SimSoC team decided to use the same technology for the SimSoC itself: the code for simulating instructions in `simlight`, i.e., the current component dedicated to the ARM v6 CPU in SimSoC, is automatically derived using a variant of our generator, whereas the previous version for ARM v5 was manually written [4].

Fig. 1 describes the overall architecture. The contributions of the work presented in this paper are the formal specification of the ARMv6 instruction set and the correctness proof of a significant operation. More precise statements on the current achievements are given in the core of the paper.

Related Work. A fully manual formalization of the fm8501 and ARMv7 architectures are reported in [9] and [7]. The formal framework is respectively ACL2 and HOL4 instead of Coq, and the target is to prove that the hardware or

[4] According to our first experiments, `simlight` compiled with Compcert is about 50 % to 70 % slower than `simlight` compiled with `gcc -O0`.

microcode implementation of ARM operations are correct wrt the ARM specification. Our work is at a different level: we want to secure the simulation of programs using ARM operations. Another major difference is the use of automatic generation from the ARM reference manual in our framework, as stated above.

The rest of the paper is organized as follows. Section 2 presents the overall architecture of `simlight` and indicates for which parts of `simlight` formal correctness is currently studied. A informal statement of our current results is also provided there. Sections 3 and 4 present respectively our Coq formal reference model of ARM and the (Coq model of) Compcert-C programs targeted for correctness. A precise statement of our current results and indications on the proofs are given in Section 5. We conclude in Section 6 with some hints on our future research directions. Some familiarity with Coq is assumed in Sections 3, 4 and 5.

2 Main Lines of SimSoC-Cert

2.1 Overall Architecture

The overall architecture of our system, called SimSoC-Cert, is given in Fig. 1. More specifically, we can see the data flow from ARMv6 Reference Manual to the simulation code. Some patches are needed from the textual version of the reference manual because the latter contains some minor bugs. 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 CompCert library which allows us, for instance, to manipulate 32-bits representations of words. The result is a set of abstract syntax trees (AST) and binary coding tables. These ASTs follow the structure of the (not formally defined) pseudo-code. Then two files are generated: a Coq file specifying the behavior of all operations (using the aforementioned Coq library) and a Compcert-C file to be linked with other components of SimSoC (each instruction can also be executed in standalone mode, for test purposes for instance). More details are provided in [4].

The decoding of ARM operations is not considered in the present paper: this is important and planned for future work, but is less urgent since we were already able to automatize the generation of intensive tests, as reported in [4]. We therefore focus first on the algorithmic body of operations. In order to state their correctness, we need Coq ASTs for the Compcert-C statements of `simlight`. The code generator directly generates such ASTs. Another option would be to work on a generated textual (ASCII) presentation of the Compcert-C code, but we prefer to avoid additional (and possibly unreliable) parsing step as far as

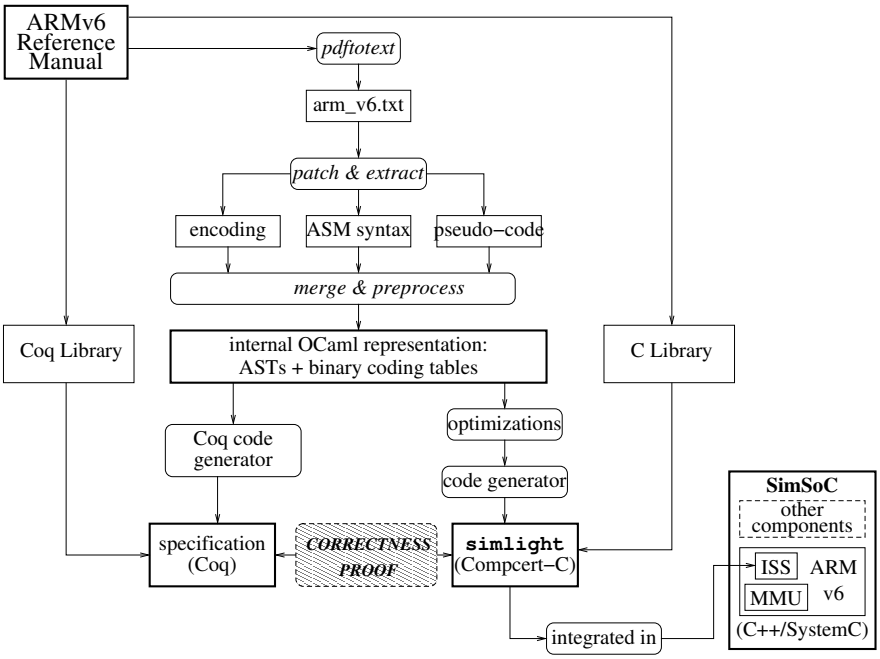

Fig. 1. Overall Architecture

possible. We will see in Section 4 that these ASTs are moreover presented in a readable form using suitable notations and auxiliary definitions.

The whole `simlight` project is currently well-compiled by Compcert (targeting Intel code) and gcc; moreover validation tests succeed completely with both simulators. The version of `simlight` compiled with Compcert can serve as a reference simulator, but for most purposes the version compiled with gcc is prefered for its higher speed.

2.2 Stating Correctness Theorems

Let us now present the purpose of the gray box of Fig. 1, which represents our main target.

The correctness of simulated ARM operations is stated with relation to the formal semantics of ARM as defined by our Coq library and partly automatically produced by the Coq code generator (the box called "specification" in Fig. 1). Note that ARM operations are presented in a readable way using suitable monadic constructs and notations: apart from the security provided by automatic generation, this greatly facilitates the comparison with the original pseudo-code of the reference manual. That said, it should be clear that the reference semantics of ARM is the Coq code provided in these files. Much effort has been spent in order to make them as clear and simple as possible.

In contrast, the Coq description of the behavior of corresponding operations (as simulated by SimSoC – Compcert-C programs) is far more complicated, though the superficial structure is quite similar. This will be detailed in Section 4. In particular, the memory model of the latter version is much more complex. In order to state correctness theorems, we define a relation between an ARM state represented by the Compcert-C memory model and another ARM state, as defined by the Coq formal semantics. Essentially, we have a projection from the former to the latter. Then for each ARM operation, we want a commutative diagram schematized in Fig. 2.

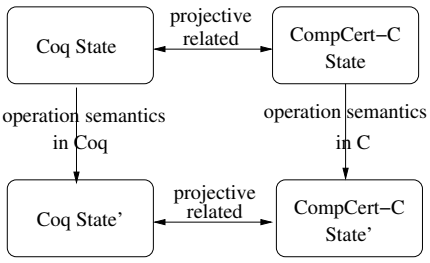

Fig. 2. Correctness of the simulation of an ARM operation

For now, our automatic generation tools operate completely, i.e., we have a Coq formal specification and a Compcert-C simulator for the full instruction set of ARM V6. About proofs, the relationship between the abstract and the concrete memory models is available; we can then state correctness theorems for all ARM operations. The work on correctness proofs themselves started recently. We considered a significant ARM operation called `ADC` (add with carry). Our main theorem (Theorem 1 in Section 5) states intuitevely that the diagram given in Fig. 2 commutes for `ADC`. Its proof is completed up to some axioms on library functions, details are given in Section 5.

3 ARM Model

3.1 Processor Behavior

A processor is essentially a transition system which operates on a state composed of registers (including the program counter) and memory. The semantics of its behavior amounts to repeat the following tasks: fetch the binary code at a given address, decode it as a processor operation and execute it; the last task includes the computation of the address of the next operation. The two main components of a processor simulator are then:

– The decoder, which, given a binary word, retrieves the name of an operation and its potential arguments.

– The precise description of transformations performed by an operation on registers and memory. In the reference manual of ARM, this is defined by an algorithm written in "pseudo-code" which calls low-level primitives for, e.g., setting a range of bits of a register to a given value. Some situations are forbidden or left unspecified. For ARM processors, this results in a so-called "UNPREDICTABLE" state. The best choice, for a simulator, is then to stop with a clear indication of what happens.

Let us illustrate this on a concrete example. Here is the original pseudo-code of the `ADC` (add with carry) operation of ARMv6. As most operations of ARM, this operation has an argument called `cond` which indicates whether the operation should be skipped or not. `CPSR` (Current Program Status Register) and `SPSR` (Saved Program Status Register, used for exception handling) are special registers related to execution modes of ARM; they also contain flags (`N`, `Z`, `C` and `V`) relevant to arithmetic instructions. The instruction has four parameters: `S` is a bit which specifies that the instruction updates `CPSR`, `Rn` is a register for the first operand, `Rd` is the destination register, `shifter_operand` specifies the second operand according to a (rather complicated) addressing mode.

```
A4.1.2 ADC
    if ConditionPassed(cond) then
        Rd = Rn + shifter_operand + C Flag;
        if S == 1 and d == 15 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);
```

In the sequel, this version of `ADC` is referred to as `ADC_pseudocode`.

3.2 Coq Semantics of ARM Operations

Each operation O from the reference manual is mechanically translated to a corresponding Coq function named O`_Coq`. First we define a type `state`, which is a record with two fields `Proc` and `SCC` (System Control Coprocessor) containing respectively the components related to the main processor (status register `CPSR`, `SPSR`, other registers...) and the corresponding components related to the coprocessor, as well as the ARM memory model. Then we use a monadic style [14] in order to take the sequentiality of transformations on the state into account. Beyond the state `st`, two other pieces of information are handled: `loc`, which represent 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 general, every O_Coq functions terminate with `Ok` as value. However for "UNPREDICTABLE" states for example, errors are implicitely propagated with our monadic constructors for exceptions : `Ko` and `Todo`.

We present now the translation of `ADC_pseudocode`. To this effect, we introduce `_get_st`, a monadic function giving access to the current state `st` in its body, represented by this notation:

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

This yields the following code for `ADC_Coq`:

```
(* A4.1.2 ADC *)
Definition ADC_Coq
   (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 st)[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 st)[Cbit]))
       ; <st> set_cpsr_bit Vbit (OverflowFrom_add3 (reg_content s0 n)
                                shifter_operand ((cpsr st)[Cbit])) ])) ]).
```

4 ARM Operations in Simlight

In the right branch of the overall architecture (Fig. 1), we generate `simlight` according to the C syntax given by Compcert. Here, actually we have two presentations of the corresponding code. The first one is in a C source which is integrated into SimSoC (see [4] for more details), its contents is:

```
/* A4.1.2 ADC */
void ADC_simlight(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);
if (ConditionPassed(&proc->cpsr, cond)) {
 set_reg_or_pc(proc,d,((old_Rn + shifter_operand) + proc->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, proc->cpsr.C_flag);
    proc->cpsr.V_flag =
     OverflowFrom_add3(old_Rn, shifter_operand, proc->cpsr.C_flag);
} } } }
```

This piece of code uses a function called `set_reg_or_pc` instead of `set_reg`: the latter also exists in `simlight` and the function to be used depends on tricky considerations about register 15, which happens to be the PC. More details about this are given in Section 4.1.

The second presentation is an AST according to a Coq inductive type defined in Compcert.

```
Definition ADC_Coq_simlight := (ADC, Internal
    {| fn_return := void; fn_params :=
      [proc -: `*` typ_SLv6_Processor;
       S -: uint8; cond -: int32; d -: uint8; n -: uint8;
       shifter_operand -: uint32];
      fn_vars := [ old_Rn -: uint32];
      fn_body :=
($ old_Rn`:o) `= (call (\reg`:o) E[\proc`:o; \n`:o] o)`:o;;
`if (• (\ConditionPassed`:o) E[&((`*(\proc`:o)`:o)|cpsr`:o)`:o; \cond`:o] o)
 then (• (\set_reg_or_pc`:o) E[\proc`:o; \d`:o; ((\old_Rn`:o)+•)+•:o] o);;
 `if ((($ S`:o)==(#1`:o)`:o)&(($ d`:o)==(#15`:o)`:o)`:o)
  then `if (call (\CurrentModeHasSPSR`:o) E[\proc`:o] o)
   then (call (\copy_StatusRegister`:o) E[&(•|cpsr`:o)`:o; •] o)
   else (call ($ unpredictable`:o) E[] o)
  else `if (($ S`:o)==(#1`:o)`:o)
   then ((($ proc`:o)|cpsr`:o)|N_flag`:o) `=
    (• (\get_bit`:o) E[(• (\reg`:o) E[\proc`:o; \d`:o] o); #31`:o] o)`:o;;
```

```
      (((($ proc‘:○)|cpsr‘:○)|Z_flag‘:○) ‘=
   (((• (\reg‘:○) • ○)==(#0‘:○)‘:○)?(#1‘:○)‘:(#0‘:○)‘:○)‘:○;;
      (((($ proc‘:○)|cpsr‘:○)|C_flag‘:○) ‘=
   (• (\CarryFrom_add3‘:○) E[•; •; (• (•|C_flag‘:○) ○)] ○)‘:○;;
      (((($ proc‘:○)|cpsr‘:○)|V_flag‘:○) ‘=
   (• (\OverflowFrom_add3‘:○) E[(• (\old_Rn‘:○) ○); •; •] ○)‘:○
  else skip
 else skip |}).
```

The symbols "○" and "•" do not belong to the real notations, they stand for types and sub-terms not represented here for the sake of simplicity. Indeed, an important practical issue is that Compcert-C ASTs include types everywhere, hence a naive approach would generates heavy and repetitive expressions at the places where ○ occurs, thus making the result unreadable (and space consuming). We therefore introduce auxiliary definitions for types and various optimizations for sharing type expressions. We also introduce additional convenient notations, as shown above for `ADC_Coq_simlight`, providing altogether a C-looking presentation of the AST.

We plan to generate the first form from the AST using a pretty-printer. The following discussion is based on the AST presentation.

4.1 Differences with the Coq Model of ARM Operations

Although the encoding of operations in `simlight` and in the Coq semantics of ARM are generated from the same pseudo-code AST, results are rather different because, on one hand, they are based on different data types and, on the other hand, their semantics operates on different memory models. Therefore, the proof that the simulation of an operation in `simlight` behaves as expected according to the Coq semantics is not trivial.

In the Coq model of ARM, everything is kept as simple as possible. ARM Registers are presented by words, the memory is a map from address to contents, the initial value of parameters such as `Rn` is available for free – we are in an functional setting, etc. In contrast, `simlight` uses an imperative setting (hence the need to store the initial value of `Rn` in `Old_Rn`, for instance). More importantly, complex and redundant data structure are involved in order to get fast speed. For example, a 32 bits wide status register is defined as a data structure containing, for every significant bit, a field of Boolean type – internally, this is represented by a byte. A more interesting example is the program counter, which is the register 15 at the same time. As this register is sometimes used as an ordinary register, and sometimes used as the PC, the corresponding data structure implemented in `simlight` includes an array which stores all the registers and a special field `pc`, which is a pointer aliasing register 15. This register plays an important role in ARM architecture. Its value is used in the `may_branch` condition for simulating basic blocks [4]. And during the simulation loop, it is read many times. Note that this special field `pc` is read-only.

Moreover we have to work with the Compcert memory model of such data structures. This, model detailed in [11], introduces unavoidable complications in

order to take low-level considerations, such as overlapping memory blocks into account. Another source of complexity is that, in a function call, a local variable takes a meaningful value only after a number of steps representing parameter binding. More details are given in Section 5.

Another important difference is that, in the Coq specification, the semantics is defined by a function wheras, in Compcert-C, the semantics is a relation between the initial memory and the final memory when evaluating statements or expressions.

4.2 Translation from Pseudo-code AST to Compcert-C AST

We describe here the mapping from the pseudo-code AST to Compcert-C AST. This translation is not only to Compcert-C AST, but more specifically to the Compcert-C AST for `simlight`. It makes use of an existing library of functions dedicated to `simlight`. For example in `ADC_pseudocode`, the occurrence of `CPSR` stands for an expression representing the contents of `CPSR` in the current state. But in `simlight`, this corresponds to a call to a library function `StatusRegister_to_uint32`. The translation deals with many similar situations.

Let us now sketch the translation process. Both the definitions of pseudo-code AST and Compcert-C AST include inductive types for expressions, statements and programs. Compcert-C expressions are limited to common programming operations like binary arithmetic operations, type cast, assignments, function calls, etc. For many constructors of pseudo-code AST the mapping is quite natural, but others require a special treatment: the ones which are specific to ARM, for representing registers, memory and coprocessor expressions, invocation of library functions, or bit range expressions. Those special expressions are translated to Compcert-C function calls. For example, the pseudo-code expression `Reg (Var n, Some m)`, designates the contents of register number `n` with the ARM processor mode `m`. In `simlight`, this becomes a call to `reg_m` with parameters `proc`, `n` and `m`.

In summary, the translation of expressions looks as follows:

```
let rec Transf_exp = function
  | Reg (e, m) -> Ecall reg_m ...
  | CPSR -> Ecall StatusRegister_to_uint32 ...
  | Memory (e, n) -> Ecall read_mem ...
  | If_exp (e1, e2, e3) -> Econdition ...
  | BinOp (e1, op, e2) -> Ebinop ...
  | Fun (f, e) -> Ecall f ...
  ...
```

For statements, we have a similar situation. Here, assignments require a special attention. For example in `ADC_pseudocode`, there is an assignment `CPSR=SPSR`. In `simlight`, this assignment is dealt with using a call to the function `copy_StatusRegister`. The corresponding Compcert-C AST embeds this call as an argument of the constructor `Sdo`.

In summary, the translation of statements looks as follows:

```
let rec Transf_stm = function
  | Assign (dst, src) -> Sdo (Ecall funct ...)
  | For (c, min, max, i) -> Sfor ...
  | If (e, i1, i2) -> Sifthenelse ...
  | Case (e, s, default) -> Sswitch ...
  ...
```

In our case, each operation is transformed to a Compcert-C program where there are no global variables, the function list contains only the function corresponding to the considered ARM operation (let us call it f, it is of course an internal function), and with an empty `main`. When the program is called, the global environment built for this program will only contain a pointer to f.

The translation from pseudo-code AST program to Compcert-C AST program has the following shape:

```
let Transformed_program =
  { vars = []
  ; functs = [ Internal (instr_id,
                 { fn_return = Tvoid
                 ; fn_params = ... (* operation parameters *)
                 ; fn_vars = ... (* operation local variables *)
                 ; fn_body = ... (Transf_stm ...) }) ]
  ; main = empty_main }
```

5 Current Proofs

On both sides, the Compcert-C `simlight` model and the Coq ARM model, the state of the processor is expressed by a big Coq term. In the Compcert-C `simlight` model, the processor state information is gathered in a data structure `SLv6_Processor`, which includes the MMU, the status registers `CPSR` and `SPSR`, the system coprocessor and the registers. In the Coq formal model of ARM, the processor state is represented by a value of type `result`, described in Section 3.2.

It is clearly possible to define a projection from a `SLv6_Processor` M to a `result` r. Then we say that M and r are *projective-related*, denoted by *proc_state_related* M r. The evil is in the details of the different type definitions, especially for the memory models. Here are the guiding ideas. Once a function such as `ADC_Coq_simlight` is called, parameters are allocated in memory, and a local environment is built. This local environment contains the mapping from identifiers to a memory block reference. For a variable of type `struct`, such as the ARM processor, the environment only yields an entry pointer to the structure. Here, the type information generated for our Compcert-C AST is needed in order to find fields inside Compcert-C memory, and to retrieve the processor model. The main function used there from Compcert is `load`. Its arguments are a memory M, a block b, an offset *ofs* and the type τ of the value to be loaded from b at *ofs*. Other variables who have a simple type like `int32`, are directly accessed by their identifier from the environment.

Let us now consider a specific instance of Fig. 2, applied to `ADC`. We choose it first because it is a typical ARM operation, which involves various ways of changing the processor state, and arithmetic calculations. Moreover, all data-processing operations have a very similar structure. If we prove the correctness of the `simlight` implementation of `ADC`, we can expect to automate the proofs for the others data-processing operations.

The proof exploits the formal operational semantics of Compcert-C, which is defined as a transition system

$$G, E \vdash \langle \text{piece of code} \rangle, M \stackrel{t}{\Rightarrow} out, M'$$

where G represents the global environment (constants) of the program, E represents the local environment, M and M' represent memory states, t is the trace of input/output events and *out* is the outcome. In our case, the piece of code is `ADC_Coq_simlight`, and the trace of input/output event (t) is empty: all function calls are internal calls. Compcert-C offers two kinds of operational semantics: small-step and big-step semantics. The latter is better suited to our needs because the statement of correctness, along the diagram in Fig. 2, relates states before and after the execution of the body of an operation. The precise statement of our theorem is as follows.

Theorem 1. *Let M and M' be the memory contents respectively before and after the simulation of `ADC_Coq_simlight`; similarly, let st and st′ be the state of ARM in its formal model. If M and st are projective-related, as well as the arguments of the call to `ADC`, then M' and st′ are projective-related as well. Formally, if:*

- *proc_state_related M (`Ok` st)*
- *similarly for the arguments of `ADC`*
- $G, E \vdash$ `ADC_Coq_simlight`$, M \stackrel{t}{\Rightarrow} out, M'$

then proc_state_related M' (`ADC_Coq` (arguments, st)).

In the Coq formal model of ARM, transitions are terminating functions returning a result of type `result`, as defined in section 3.

The proof process is driven by the structure of the operation body. Step by step, we observe the memory changes on the Compcert-C side and the state changes on the Coq side, and we check whether the relation still holds between the current Compcert-C memory state and the Coq state. To this effect, we apply theorems on load/store functions from Compcert [11]. Proof by computation does not work because the types involved are complex – they embed logical information – and many definitions are opaque.

In `ADC_Coq`, conditional expressions and function calls for getting values have no side effect on the state. On the Compcert-C side, declaring a local variable in a function has no impact on the memory model of the processor. The state may only change when a function for setting values is called, like `set_reg`, `copy_StatusRegister`, or assignment of bits in register fields. Such calls will

return a new memory state on the Compcert-C side and a new `Ok` state on the Coq side. We use small-step semantics for such steps.

Now we need some lemmas for these proof steps. Lemmas can be organized into four kinds. We give an instance of each kind.

Lemma 1. *The conditional expression `S==1` has no effect on Compcert-C memory state:*
if $G, E \vdash \texttt{condition}_C?a1 : a2, M \overset{E0}{\Rightarrow} vres, M'$,
then M=M'.

Lemma 1 is easy to prove by some inversions. All lemmas of this kind have been discharged.

Lemma 2. *The conditional expression `S==1` has the same result in the Compcert-C model as in the Coq model:*
if $G, E \vdash \texttt{condition}_C?a1 : a2, M \overset{E0}{\Rightarrow} vres, M'$*:*
- and if $is\_true\ vres$*, then* $condition_{Coq} = true$
- and if $is\_false\ vres$*, then* $condition_{Coq} = false$.

To prove lemma 2, we need to apply small-step semantics, to check the type of `S` and the value of the Boolean result `vres`. Note that in Compcert-C, non-zero integer, non-zero floats and non-null pointer can be interpreted as the Boolean value true, which adds some complexity in the proof.

The proof is by case analysis according to the type of `vres`. As the expression involves a parameter (S), the projective relation about this parameter between Compcert-C memory and the formal model of ARM is required.

All lemmas of this kind have been discharged.

A lemma of the two next kinds is stated for each `simlight` library function which changes the state, e.g., `set_reg`.

Lemma 3. *If proc_state_related* M (`Ok` st),
and if $G, E \vdash \texttt{set\_reg}_c(proc, reg\_id, data), M \overset{E0}{\Rightarrow} vres, M'$,
then proc_state_related M' ($\texttt{set\_reg}_{Coq}$ st).

For the moment, such lemmas are considered as axioms on the library. In order to be properly stated, we need the Compcert-C ASTs of such library functions, which are not automatically generated. We have 6 lemmas/axioms of this kind for `ADC`.

The next lemma is stated for a given call to `set_reg` in the body of the function `ADC_Coq_simlight` and a parameter P of `ADC_Coq_simlight` which is not used as an argument of `set_reg`.

Lemma 4. *After the call to `set_reg`, the value of P remains unchanged:*
if $G, E \vdash \texttt{set\_reg}_c(proc, reg\_id, data), M \overset{E0}{\Rightarrow} vres, M'$,
then $P(M) = P(M')$.

Lemma 4 can be proved with the help of theorems of Compcert on "load after store". A typical proof step looks like:

Original ARM ref man (txt)	49655
ARM Parsing to an OCaml AST	1068
Generator (Simgen) for ARM and SH with OCaml and Coq pretty-printers	10675
Generated C code for Simlight ARM operations	6681
General Coq libraries on ARM	1569
Proof script on `ADC`	1461

Table 1. Sizes (in number of lines)

> If we store a value v on block b ($\texttt{store}(M1, \tau, b, ofs, v) = M2$), then the contents of block b' remains unchanged ($\texttt{load}(\tau', M2, b', ofs') = \texttt{load}(\tau', M1, b', ofs')$) for any type τ' and offset ofs', which makes the accesses disjoint ($b' \neq b$ or $ofs + |\tau| \leq ofs'$ or $ofs' + |\tau'| \leq ofs$).

As for lemmas 3, we need additional axioms on `simlight` library functions.

Our current result is that, with the help of these lemmas, we have a complete correctness proof for `ADC` (Theorem 1). Theorem 1 and all the lemmas are in the file `correctness_ADC.v`[5].

The whole proof structure of this theorem and all twenty lemmas of kinds 1 and 2 were completed within 2 weeks. The 10 remaining lemmas, of kinds 3 and 4, should require a similar effort. Here, we first need to generate Compcert-C ASTs for the relevant library functions using the C parser available in Compcert.

6 Conclusion

The trust we may have in our result depends on the faithfulness of its statement with relation to the expected behavior of the simulation of `ADC` in `simlight`. It is mainly based on the manually written Coq and C library functions, the translators written in OCaml described in Section 2 (including the pretty-printer for Coq), the final phase of the Compcert compiler, and the formal definition of *proc_state_related*.

The current development is available online[5]. Figures on the size of our current development are given in Table 1.

In the near future, we will extend the work done on `ADC` to all other operations. The first step will be to design relevant suitable tactics, from our experience on `ADC`, in order to shorten a lot the current proof and make it easier to handle and to generalize. We are confident that the corresponding work on the remaining ARM operations will then be done much faster, at least for arithmetical and Boolean operations.

Later on, we will consider similar proofs for the decoder – as for the body of operations, it is already automatically extracted from the ARM reference manual. Then a proven simulation loop (basically, repeat decoding and running operations) will be within reach.

[5] `http://formes.asia/media/simsoc-cert/`

In another direction, we also reuse the methodology based on automatic generation of simulation code and Coq specification for other processors. The next one which is already considered is SH4. In fact, the same approach as the ARMv6 has been followed, and a similar Coq representation can currently be generated from the SH4 manual. Moreover, as the SH pseudo-code is simpler than the ARM, we are hence impatient to work on its equivalence proof.

Acknowledgement

We are grateful to Vania Joloboff and Claude Helmstetter for their many explanations on SimSoC. We also wish to thank the anonymous reviewers for their detailed comments and questions.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. Coq Development Team. *The Coq Reference Manual, Version 8.2*. INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`.
6. L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles. *Frama-C User Manual, Release Boron-20100401*. CEA LIST, Software Reliability Laboratory, Saclay, France, 2010.
7. A. C. J. Fox and M. O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP*, pages 243–258, 2010.
8. C. Helmstetter, V. Joloboff, and H. Xiao. SimSoC: A full system simulation software for embedded systems. In IEEE, editor, *OSSC'09*, 2009.
9. W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNAI*. Springer-Verlag, 1994.
10. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
11. 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.
12. P. S. Magnusson and al. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
13. Open SystemC Initiative. *SystemC v2.2.0 Language Reference Manual (IEEE Std 1666-2005)*, 2006. `http://www.systemc.org/`.
14. S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, 2010.
15. OSCI SystemC TLM 2.0.1, 2007. `http://www.systemc.org/`.