ARMY: A deductive verification platform for ARM programs using Why3 ⋆

Mário Pereira[1], Jean-Christophe Filliâtre[2,3], and Simão Melo de Sousa[1]

[1] {a25270|desousa}@ubi.pt
LIACC & DI, University of Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001 Covilhã, Portugal
[2] Jean-Christophe.Filliatre@lri.fr
Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[3] INRIA Saclay - Île-de-France, Orsay, F-91893

Abstract. Unstructured (low-level) programs tend to be challenging to prove correct, since the control flow is arbitrary complex and there are no obvious points in the code where to insert logical assertions. In this paper, we present a system to formally verify ARM programs, based on a flow sequentialization methodology and a formalized ARM semantics. This system, built upon the Why3 verification platform, takes an annotated ARM program, turns it into a set of purely sequential flow programs, translates these programs' instructions into the corresponding formalized opcodes and finally calls the Why3 VCGen to generate the verification conditions that can then be discharged by provers. A prototype has been implemented and used to verify several programming examples.

1 Introduction

The process of deductive software proving [9] is somehow routined when it comes to prove programs with structured control flow graphs: we annotate functions with pre- and postconditions and loops with invariants and variants and then we rely on the VCGens and theorem provers (either automatic or interactive) to prove the validity of the computed verification conditions. If the verification conditions are all discharged then the program is guaranteed to be correct.

Regarding unstructured programs the verification process is not that simple: due to its unstructured nature, the control flow graph of these programs does not favor any point to insert logical annotations. For these cases traditional verification techniques are useless.

In this paper we present our work regarding proofs of unstructured programs. Our objective is to augment the Why3 platform [3,4] with the ability of proving the correctness of ARM [14] programs. To accomplish this goal we apply a tool supported methodology to turn a program code into a set of purely sequential programs (eliminating the unstructured nature of the program control-flow

⋆ This work was partially funded by a Research Grant from the RELEASE laboratory, University of Beira Interior, and by Fundação para a Ciência e Tecnologia in the FAVAS research project FCT-PTDC/EIA-CCO/105034/2008.

graph) whose correctness implies the correctness of the initial program. We use WhyML, the programming language of Why3, to encode the semantics (of a subset for now) of the ARM assembly language. This subset includes all the opcodes required to encode usual programs and algorithms.

The system we propose to prove ARM programs has the following working-pipeline: (1) an annotated ARM source code is parsed in order to create its control flow graph and, from that, a set of purely sequential programs; (2) the sequential programs are translated into the syntax of the formalized opcodes (WhyML syntax); (3) finally, Why3 VCGen is used to compute verification conditions and proofs can be conducted over them.

We present a set of small-step operational semantics rules used to formalize ARM opcodes. Another contribution of this work is the Why3 module we created to model ARM opcodes. We also describe a prototype tool we conceived to automate all the verification process and to produce the practical results of our work.

This paper is organized as follows. Section 2 presents some previous contributions related to our work. Sections 3 and 4 serve as overviews of Why3 and ARM architecture, respectively. Section 5 describes the core of our contribution: the formalization of the ARM opcodes semantics and the corresponding Why3 module. Section 6 explains the sequentialization methodology. Section 7 illustrates some case studies. Finally, we conclude with some final remarks and possible future work.

2 Related Work

Barnett and Leino present an approach [2] which takes advantage of the Spec# [1] programming system to interpret a general-control flow graph. The multiply entry points to loops are eliminated from this graph. Then, loops are eliminated producing an acyclic control-flow graph and a single-assignment transformation is applied to the acyclic graph by changing all the assignment statements into assume ones. Finally, the weakest pre-condition calculus [7] is used to compute the verification conditions.

Filliâtre [8] chooses to tackle the issue of assertions location by giving complete freedom about their location within the code: assertions can be inserted at any point of the program, the sole restriction being that any cycle must contain at least one assertion. The next step of this methodology is to turn the assembly program into a set of purely sequential programs, that is with no jump instructions. In the end, a traditional weakest pre-condition calculus is used to infer the verification conditions of every sequence. The correctness of individual sequences implies the correctness of the initial program.

Regarding hardware verification, the ARM processor family has been the subject of several case studies reported in the literature. Fox and Myreen used the HOL4 proof assistant [16] to model the ARMv6 and ARMv7 processors and respective Instruction Sets [10,11]. These works attempt to formally verify the functional correctness of the processors and also to specify and verify machine

code. So, HOL4 presents itself as the appropriate ground to achieve these goals, since it is a proof system projected to perform hardware proving.

The simulation of Systems-on-Chip plays an important role when developing for specific boards, such as ARM CPUs. It is of major importance to be able to guarantee a safe and reliable simulation of the created programs before these are shifted to the real hardware. Shi et al [15] model the ARM architecture in the Coq proof assistant [6] and consider parts of the SimSoC simulator [12] written in Compcert-C [13]. Using the Compcert-C formal semantics they proved that the simulation of ARM operations conforms to their conceived formal model of ARM processor.

3 Why3 Overview

Why3 is a platform for program verification [3,4]. It is mainly composed of two parts: the logical side, which presents the Why language and is equipped with a mechanism to translate this language to existing theorems provers; and a programming language, WhyML, with a verification condition generator.

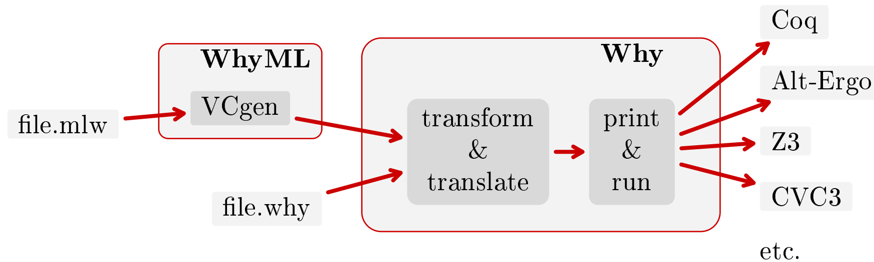

Fig. 1. Why3 workflow

Why3 presents a polymorphic first-order logic extended with algebraic data types and inductive predicates [5]. Small units can be used to aggregate logical declarations, called *theories* (a .why file). Why3's standard library is composed of several of these theories, some of them being of major importance for program verification (integer arithmetic, sets, maps).

WhyML is a first-order language with a ML-like syntax providing the common constructs of imperative programming languages (loops, exceptions, references, sequences) as well as ML constructs (pattern matching, local functions, polymorphism). In WhyML, one can prove behavioral correctness, safety properties, and termination, through the insertion of logical annotations. Similarly to logical declarations, program declarations can be grouped together in *modules* (in `.mlw` files). Why3 standard library provides some important modules for program verification, *e.g.* arrays and references.

As aforementioned, program properties (behavioral correctness, safety, termination) are proved within Why3 by annotating the code with first-order logical assertions. Pre-contitions are used to state the program initial requirements;

postconditions refer to the expected result or final state of variables; loop invariants state properties that stay unchanged during loop execution; finally loop variants are used to ensure termination (although being the most common type used to state loop variants, Why3 does not limit variants to natural numbers, accepting instead any type equipped with a well-founded order relation).

Using a weakest precondition calculus, verification conditions (VC for short) are computed from the logical assertions inserted in the code. Finally, Why3's back-end sends these verification conditions to theorem provers, with their validity implying the validity of the program.

4 ARM in a Nutshell

ARM architecture is a common 32-bit processors architecture found in many embedded and mobile systems. This architecture family offers high performance for limited power devices as well as small ones.

This architecture presents a three-stage pipeline for instruction execution: fetch, decode and execute. This pipeline organization allows an increase in the speed of the flow of instructions to the processor.

Thirty-seven registers are available within ARM processors, 31 of those are general-purpose 32-bit registers and 6 status registers(1 CPSR, Current Program Status Register, and 5 SPSR, Saved Program Status Register). The accessibility of these registers to the programmer depends on the processor state and operating mode.

The following small example represents a piece of an ARM program that, by modifying and later reading the contents of the CPSR, performs (or not) a data-processing operation:

```
mov r0, r3
cmp r0, r1
addge r0, r0, r2
```

This program starts by passing the value contained in register `R3` to register `R0` and then compares its value with the value stored in register `R1`. The compare (`cmp` opcode) will modify the value of the CPSR to express the result of the comparison performed. Finally, only if the value of `R0` is greater or equal to the value of `R1` the last instruction will have effect (adding to the value `R0` the value of `R2`).

5 ARM Encoding In Why3

5.1 Opcodes Selection

ARM offers a comprehensive set of instructions, many of which are not frequently used. Our first approach in formalizing the ARM Instruction Set was not to encode all instructions of this architecture. Instead, we used a C to ARM7TDMI cross-compiler [17] to generate the Assembly representations of a set of representative programming examples, and so understand which are the opcodes that are necessary to formalize algorithms and programs.

The following are the necessary opcodes to prove the correctness of the case studies presented in this paper:

– MOV{cond} ⟨rd⟩ ⟨shifter_operand⟩: performs a move operation of the value of **shifter_operand** to register **rd**;
– STR{cond} ⟨rd⟩ ⟨addressing_mode⟩: performs a store operation of the **rd** register value to the memory location specified by **addressing_mode**;
– LDR{cond} ⟨rd⟩ ⟨addressing_mode⟩: performs a load operation of the value stored in **addressing_mode** memory location to register **rd**;
– ADD{cond} ⟨rd⟩, ⟨rn⟩, ⟨shifter_operand⟩: performs an addition operation of register **rn** value and **shifter_operand** value to register **rd**;
– SUB{cond} ⟨rd⟩, ⟨rn⟩, ⟨shifter_operand⟩: performs a subtraction operation of **rd** register value and **shifter_operand** value to register **rd**;
– RSB{cond} ⟨rd⟩, ⟨rn⟩, ⟨shifter_operand⟩: performs a reserve subtraction operation of **rn** register value and **shifter_operand** value to register **rd**;
– MUL{cond} ⟨rd⟩, ⟨rn⟩, ⟨rs⟩: performs a multiplication operation of registers **rn** and **rs** values to register **rd**;
– CMP{cond} ⟨rd⟩, ⟨shifter_operand⟩, performs a comparison operation of **rd** register value and **shifter_operand** value.

As mentioned, these eight opcodes represent the subset of the ARM Instruction Set necessary to cover all the instructions from the selected programs. Consequently, encoding these opcodes will suffice to verify the same selected programs. It is worth pointing out that if future programs present non formalized opcodes, these can be easily added to our formalization.

5.2 Opcodes Semantics

To expose the syntax of the selected opcodes we devised a minimal grammar (figure 2) [4], serving as base for the embedding of these instructions on the Why3 platform.

⟨**r**⟩::= R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | FP | R12 | SP | LR | PC
⟨**op**⟩::= mov | str | ldr | add | sub | rsb | mul | cmp
⟨**c**⟩::= EQ | NE | CS | HS | CC | LO | MI | PL | VS | VC | HI | LS | GE | LT | GT | LE | AL
⟨**i**⟩::= #*n*   ⟨**sop**⟩::= ⟨r⟩ | ⟨i⟩   ⟨**addm**⟩::= [⟨r_1⟩, ⟨r_2⟩] | [⟨r⟩, ⟨i⟩]
⟨**inst**⟩::= ⟨op⟩⟨c⟩⟨r⟩list⟨sop⟩ | ⟨op⟩⟨c⟩⟨r⟩list | ⟨op⟩⟨c⟩⟨r⟩⟨addm⟩

Fig. 2. Grammar for the ARM selected instructions

In this grammar we present the 16 registers possibly used by the ARM processor, from R0 to R15, which happens to be the *program counter*; the 17 possible

[4] As it can be noticed, branching instructions are not included in the grammar rules. This happens due to the fact that sequentialization mechanism removes those instructions from the code of sequential programs.

conditional execution values, since most ARM instructions can be conditionally executed, meaning they will only have effect according to CPSR (Current Program Status Register) state, especially if the N, Z, C and V flags satisfy the conditional value specified; the syntax for immediate values, a '#' followed by a signed 32 bits integer (n stands for signed-integers); two possible addressing modes for instructions operands, shifter operands (used for data-processing operations) and addressing mode for load/store operations (respectively the **sop** and the **addm** grammatical classes); and finally we give a general syntax for an instruction, an opcode with a condition value followed by a list of registers and possible addressing modes.

$$[\text{mov}] \quad \frac{}{\langle \mathbf{mov}\, c\, r\, sop, s\rangle \rightsquigarrow s[r \mapsto \mathcal{S}\,[\![sop]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{str}] \quad \frac{}{\langle \mathbf{str}\, c\, r\, addm, s\rangle \rightsquigarrow s[\text{Memory}[\mathcal{A}\,[\![addm]\!]\, s] \mapsto \mathcal{R}\,[\![r]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{ldr}] \quad \frac{}{\langle \mathbf{ldr}\, c\, r\, addm, s\rangle \rightsquigarrow s[r \mapsto \text{Memory}[\mathcal{A}\,[\![addm]\!]\, s]]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{add}] \quad \frac{}{\langle \mathbf{add}\, c\, r_1\, r_2\, sop, s\rangle \rightsquigarrow s[r1 \mapsto \mathcal{R}\,[\![r_2]\!]\, s + \mathcal{S}\,[\![sop]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{sub}] \quad \frac{}{\langle \mathbf{sub}\, c\, r_1\, r_2\, sop, s\rangle \rightsquigarrow s[r1 \mapsto \mathcal{R}\,[\![r_2]\!]\, s - \mathcal{S}\,[\![sop]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{rsb}] \quad \frac{}{\langle \mathbf{rsb}\, c\, r_1\, r_2\, sop, s\rangle \rightsquigarrow s[r1 \mapsto \mathcal{S}\,[\![sop]\!]\, s - \mathcal{R}\,[\![r_2]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{mul}] \quad \frac{}{\langle \mathbf{mul}\, c\, r_1\, r_2\, r_3, s\rangle \rightsquigarrow s[r1 \mapsto \mathcal{R}\,[\![r_2]\!]\, s \cdot \mathcal{R}\,[\![r_3]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{cmp}] \quad \frac{}{\langle \mathbf{cmp}\, c\, r\, sop, s\rangle \rightsquigarrow s[cmp_result \mapsto \mathcal{R}\,[\![r]\!]\, s - \mathcal{S}\,[\![sop]\!]\, s]} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{tt}$$

$$[\text{nop}] \quad \frac{}{\langle \mathbf{x}, s\rangle \rightsquigarrow s} \quad \text{if } C\,[\![c]\!]\, s = \mathbf{ff}, \forall x \in \text{inst}$$

Fig. 3. Small-step semantics rules for ARM selected opcodes

To formalize the operational semantics of the established language subset we used a traditional small-step semantics (figure 3). A program state describes the content of the registers, CPSR, memory and a global variable **cmp_result** representing the result of a comparison operation. In our rules this program state is represented by s.

Instruction semantics is given by a relation of the form $\langle inst, s\rangle \rightsquigarrow s'$, with $inst$ representing the instruction being executed, s representing the program state from which the instruction begins its execution and s' representing the program state after the execution of the instruction. In our rules s' is always represented using the notation $s[\alpha \mapsto \gamma]$. This means that s' is a state equivalent to s except that the value bound to α is now the value of γ.

To determine if an instruction may execute we defined the function $\mathcal{C}[\![\cdot]\!]$ that takes as parameters a condition and a state: $\mathcal{C} : c \rightarrow s \rightarrow T$. This semantic function means that to a given condition and a program state it evaluates the state of the CPSR and returns a value of type T (T consists of the truth values

tt for true and **ff** for false) indicating whether or not the CPSR state satisfies the conditional value passed. The last rule formalizes the situation when $\mathcal{C}[\![\cdot]\!]$ returns **ff**, in which all instructions behave as `nop`, that is, not altering the program state.

The value of a register at any program state is given by the function $\mathcal{R}[\![\cdot]\!]$ with signature $\mathcal{R} : r \to s \to n$ indicating that for a register name and a program state this function returns an integer value (n, as well as for the grammar production rules, here stands for signed-integers values).

Regarding load and store operations, the value of the addressing mode for these operations is given by the function $\mathcal{A}[\![\cdot]\!]$ with signature $\mathcal{A} : addm \to s \to n$. This function receives an addressing mode (represented by the value $addm$) and a state and returns an integer value indicating the desired memory region (the memory is represented in the semantic rules by the value `Memory`).

Finally, the value of a shifter operand for normal data-processing operations (the value sop in the semantic rules) is given by the function $\mathcal{S}[\![\cdot]\!]$: $\mathcal{S} : sop \to s \to n$. The signature of this function indicates that it receives a shifter operand (the value sop) and a state, returning an integer value representing the value of the shifter operand.

The semantics for a sequence of instructions is the usual one: the state resulting from first instruction execution is the state starting the execution of the remaining ones.

5.3 Opcodes Encoding in Why3

To be able to apply the Why3 VCGen engine to ARM Assembly programs we conceived a logical model to represent these programs: (1) the registers are modeled as a global map indexed from a type *register* to int32 (this modelling can be easily extended to a bit-vector structure to conform with lower-level issues, namely the value of individual bits); (2) memory is modeled as a global map from integer to int32; (3) CPSR is modeled as global map from integer to int32; (4) the result of a comparison is stored in a global integer variable, *cmp_result*.

In order to encode the operation of each opcode we used the Why3 procedures' syntax. The effect of Why3 procedures is presented within the body of these procedures, indicating the set of variables whose value is read or written by the procedure. Also in the body the return value type of the procedure must be specified. The expected behaviour of the procedure is stated in its post-condition.

To make our ARM model and respective procedures accessible to further use we created a Why3 module with the name **ARM** enclosed in the **arm.mlw** file. By importing this module one can have access to all the encoded opcodes, as well as to the modeled memory, CPSR or registers bank.

6 Sequentialization Process

In this section we present the methodology used to prove the ARM Assembly programs. The contents presented in this section are mainly based in those pre-

sented in an unpublished work on proving MIX programs [8], by one of the authors.

6.1 Sequentialization

To overcome the problems associated with the unstructured nature of Assembly programs control flow graph we adopted a methodology where any Assembly program is turned into a set of purely sequential programs. The resulting sequential programs do not include any branch instructions, either conditional or not. In the end, if all these programs are proved to be correct than the initial program is also correct.

The sequentialization process starts from constructing the control flow graph of the program, where nodes are program points and transitions are sequences of instructions. In this control flow graph, every loop must contain at least one logical annotation that works as an invariant: this annotation must be verified when reached initially (loop invariant initialization) and its validity must be maintained by any path in the graph reaching the annotation point again.

To perform the program sequentialization a deep-first traversal algorithm is used, starting from the graph entry point and associating to each node a set of purely sequential programs. By the end of the traversal the resulting set of programs is the set of purely sequential programs associated to the entry point and to each invariant that may arise during the traversal. For a pseudo-code for this algorithm please refer to [8].

6.2 Methodology Soundness

The set of resulting sequential programs is a set of programs with shape

assume P
s
assert Q

where s is a purely sequential program, the keyword **assume** introduces an initial assumption and **assert** a logical property that must be verified. Let us assume a transition relation for purely sequential programs denoted by $S_1 \xrightarrow{s} S_2$ meaning that execution of program s from state S_1 will lead to state S_2. Based on this transition and in the shape of each sequential program, proving the soundness of the sequentialization method (and consequently the correctness of each program) means to prove that for a program s starting in a state S_1 satisfying P (noted $S_1 \vDash P$) and for a state S_2 such that $S_1 \xrightarrow{s} S_2$ we have $S_2 \vDash Q$.

Theorem 1 (soundness). *Let S be a state satisfying the invariant I at entry point, i.e. $S \vDash I$, and let us consider an execution reaching a program point with an invariant J in state S'. Then $S' \vDash J$.*

Proof. For a complete proof on the soundness of sequentialization algorithm please refer to [8].

7 Case Studies

A prototype tool has been implemented in order to practically test the methodologies proposed in this work, the **army** tool (ARM in whY). This tool serves as a front-end to the Why3 platform, accepting as input annotated ARM programs, converting these programs into a set of purely sequential programs and finally translating the resulting sequential programs to the WhyML syntax. The ARM instructions will be mapped to the respective procedure of the created Why3 **ARM** module and so allowing the Why3 VCGen to generate proof obligations implying the validity of the ARM code. These proof obligations can then be discharged by theorem provers (Coq, Z3, Alt-ergo, Simplify).

The **army** tool takes a .s file with ARM code logically annotated and produces a .mlw file containing the translation of the original code to the WhyML syntax. In the .s file annotations are introduced via three different environments: (i) assertions are enclosed by @a ... a@; (ii) an invariant is introduced within @i ... i@; (iii) @v ... v@ is used to directly insert Why3 code into the .s file.

7.1 Euclidean Division

We present the proof of a program computing the quotient and the remainder of the division between two positive natural numbers using the Euclidean Division. To prove the correctness of this program one must prove that $x = q * y + r$, $0 \leq r < y$ with q and r being the final calculated quotient and remainder (respectively) and x and y the divided numbers.

We assume that register R0 contains the value x, R1 contains the value of y, R2 contains the value of an auxiliary variable a and R3 and R4 will contain the final values for q and r, respectively. The annotated code is as follows:

```
div:      @i!registers[R0] > 0 ∧ !registers[R1] > 0 i@
          mov r2, #1
          mov r3, #0
          mov r4, #0
          b .L2
.L5:
          sub r5, r1, #1
          cmp r4, r5
          bne .L3
          add r3, r3, #1
          mov r4, #0
          b .L4
.L3:
          add r4, r4, #1
.L4:
          add r2, r2, #1
.L2:      @i
          (0 ≤ !registers[R4] < !registers[R1]) ∧
          (1 ≤ !registers[R2] ≤ !registers[R0] + 1) ∧
          (!registers[R2]=!registers[R1]*!registers[R3]+!registers[R4]+1)
          i@
          cmp r2, r0
          ble .L5
          @a
          (!registers[R0]=!registers[R3]*!registers[R1]+!registers[R4]) ∧
          (0 ≤ !registers[R4] < !registers[R1])
          a@
```

The pre-condition for this program is inserted right at the entry point (label **div**), stating the values of registers R0 and R1 must be both positive at the start of the program.

The invariant is here located at the loop entry point (label **.L2**). This invariant indicates that the value of register R4 should be between 0 and the value of register R1 ($0 < r < y$), indicates that the value of register R2 should be greater or equal to 1 and less or equal to the value of register R0 plus one ($1 \leq a \leq x+1$) and finally states the property $a = y \times q + r + 1$ (the value of register R2 is equal to the value of register R1 times the value of register R3 plus the value of register R4 plus one). The meaning of this annotation is straightforward: every state of execution reaching it must satisfy its properties.

The postcondition states the expected result, $x = y * q + r$, through the expression **!registers[R0]** = **!registers[R3] * !registers[R1]** + **!registers[R4]** and also indicates the remainder of the division is greater or equal to 0 and less than the divisor ($0 \leq r < y$).

Running this program into **army** tool will generate four sequential programs and these will generate a total of 18 verification conditions: 5 related with sequence one; 5 related with sequence two; 3 related with sequence 3; and 5 related with sequence 4. All of these verification conditions are automatically discharged using Z3, CVC3 and Alt-Ergo provers.

7.2 Array search

The next example illustrates the proof of a program searching for the first occurrence of an element within an array. The annotated code is the following:

```
search: @i !registers[R1] ≥ 1 i@
        mov r2, #0
        b .L2
.L4:
        mov r0, r2
        add r2, r0, #1
.L2:    @i
        (0 ≤ !registers[R2] ≤ !registers[R1]) ∧
        (∀ i:int. 0 ≤ i < !registers[R2] → !memory[i] ≠ !registers[R3])
        i@
        cmp r2, r1
        bge .L3
        ldr r0, [r2, #0]
        cmp r0, r3
        bne .L4
.L3:
        mov r0, r2
        @a
        (!memory[!registers[R0]] = !registers[R3] ∧
        ∀ i:int. 0 ≤ i < !registers[R0] → !memory[i] ≠ !registers[R3])
        ∨
        (∀ i:int. 0 ≤ i < !registers[R1] → !memory[i] ≠ !registers[R3])
        a@
```

For this program we assume that register R3 holds the value to be searched for; R1 contains the size of the array; and the elements of the array are stored in memory at addresses $0, \cdots, R1-1$. During the program execution R2 is used to scan the array and when the program returns R0 will hold the index of the

searched value (if this exists in the array). The postcondition presents a disjunction, indicating the possibility of a successful search (the elements is actually contained within the bounds of the array) or a unsuccessful one (the desired element does not exist in the array). The six verification conditions generated for this case are discharged fully automatically.

8 Conclusion and Future Work

In this paper we have presented our contribution regarding the proof of unstructured programs, having ARM Assembly as the target language. We described a complete framework to prove the correctness of such programs, starting from annotated ARM programs and ending with a set of verification conditions generated and handled by the Why3's VCGen. We exposed two practical examples of our work, serving as proofs of concept for the formalizations and methodologies reported in this document.

The first experiments have shown motivating results, and so we are considering to attend in more detail the verification of ARM programs. Our nearest extension will be to formally reason about programs complexity, by augmenting the opcodes with an execution cost and building a mechanism to check the complexity of a program from its instructions against an expected overall cost. This addition to our contribution will force us to reason in a more detailed fashion about 32-bit machine numbers, since the time cost of some instructions is strictly dependent of the arithmetic properties of its 32-bit operands. For this purpose we must refine our semantics in order to cope with a bit level representation of register values (as provided by the Why3's bit-vector theory) and a finer model of the opcodes' semantics.

As mentioned before, the ARM processors present a three-stages pipeline. So, it would be of much interest to refine the encoded semantics to consider formal proofs taking into account the flow of instructions in this pipeline. An other feature of the processor omitted from this work and that it could be also of much interest is the cache behaviour (which will contemplate the ARM710T core). To reason about the way programs interact with the processor cache would allow us to have a much more precise mechanism to conduct proofs over ARM programs, even with the possibility of addressing complex programs safety issues.

The presented extensions to our work will introduce a much more refined model of the ARM processor and Instruction Set. This will allow to reason about how the code behaves in terms of CPU cycles. This implies 32-bit numbers, pipeline and cache to be included in our future model. Then, it will be possible to state complex programs features, such as the Worst Case Execution Time (WCET) [18].

References

1. Mike Barnett, K. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet,

and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30569-9_3.

2. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, September 2005.
3. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform, version 0.72*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.72 edition, May 2012. `https://gforge.inria.fr/docman/view.php/2990/7919/manual-0.72.pdf`.
4. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.
5. François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, volume 6989, 2011.
6. Coq Development Team. *The Coq Reference Manual, Version 8.3pl3*. INRIA Rocquencourt, France, 2011.
7. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.
8. Jean-Christophe Filliâtre. Formal Verification of MIX Programs. In *Journées en l'honneur de Donald E. Knuth*, Bordeaux, France, October 2007.
9. Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):397–403, August 2011.
10. Anthony Fox. ARM formalized in HOL. `http://testandverification.com/files/AnthonyFox_ISA_modelling.pdf`.
11. Anthony Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In *In Proc. 23rd Int. Conf on Interactive Theorem Proving (ITP'10), LNCS*. Springer, 2010.
12. Claude Helmstetter, Vania Joloboff, and Hui Xiao. SimSoC: A full system simulation software for embedded systems. In IEEE, editor, *2009 International Workshop on Open-source Software for Scientific Computation (OSSC-2009)*, page 7 p., Guiyang, Chine, September 2009. LIAMA, Institute of Automation, CAS, Beijing, China Guizhou Normal University, China. ossc2009 ossc2009.
13. Xavier Leroy. The CompCert verified compiler, software and commented proof, March 2011.
14. ARM Limited. *ARM Architecture Reference Manual*, 2005.
15. Xiaomu Shi, Jean-François Monin, Frédéric Tuong, and Frédéric Blanqui. First steps towards the certification of an arm simulator using compcert. *CoRR*, abs/1202.6472, 2012.
16. Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, pages 28–32, 2008.
17. Sourcery CodeBench Tools. `http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/`.
18. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.