Hoare Logic for ARM Machine Code

Magnus O. Myreen, Anthony C. J. Fox, Michael J. C. Gordon

Computer Laboratory, University of Cambridge, Cambridge, UK

Abstract. This paper shows how a machine-code Hoare logic is used to lift reasoning from the tedious operational model of a machine language to a manageable level of abstraction without making simplifying assumptions. A Hoare logic is placed on top of a high-fidelity model of the ARM instruction set. We show how the generality of ARM instructions is captured by specifications in the logic and how the logic can be used to prove loops and procedures that traverse pointer-based data structures. The presented work has been mechanised in the HOL4 theorem prover and is currently being used to verify ARM machine code implementations of arithmetic and cryptographic operations.

1 Introduction

Although software runs on real machines like Intel, AMD, Sun, IBM, HP and ARM processors, most current verification activity is performed using highly simplified abstract models. For bug finding this is sensible, as simple models are much more tractable than realistic models. However, the use of unrealistically simple models is unsatisfactory for assurance of correctness, since correctness-critical low level details will not have been taken into account. Details that are frequently overlooked at the low levels include: finiteness of stacks and integers, whether or not addresses need to be aligned and details of status bits.

Recently software verification based on realistically modelled software has received an increasing amount of attention as tools become able to cope with tedious operational models. Boyer and Yu [1] have done some impressive pioneering work on verification of programs for the Motorola MC68020, Tan and Appel [7] verified memory safety of Sun's SPARC machine code, and Hardin *et al.* [4] verified machine code written for Rockwell Collins AAMP7G.

Curiously these efforts have made little use of advances in programming logics, while efforts for proving programs written in realistically modelled low-level programming languages such as C have [8]. The work of Tan and Appel is – to the best of our knowledge – the only significant effort that places a general programming logic on top of a realistically modelled machine language. Their approach requires substantial effort to prove the soundness of applying the logic to their SPARC model. Hardin *et al.* and Boyer and Yu verify machine-code programs using a form of symbolic simulation of the bare operational semantics of their respective processor models.

In an earlier paper we developed a general Hoare logic for realistically modelled machine code [5]. In this paper the general logic is specialised to a detailed

model of ARM machine code. This paper shows how the logic captures the details of ARM instructions and uses examples to illustrate how programs can be proved using the logic. The examples present proofs of loops and procedures that traverse recursive data structures.

This paper avoids a lengthy proof of soundness by simply instantiating abbreviating definitions for which sound proof rules have been proved in an earlier paper [5]. All specifications and proofs presented in this paper have been mechanically checked using the HOL4 system [3]. The detailed ARM model at the base of this work has been extracted from a proof of correctness of the instruction set architecture of an ARM processor [2].

The remainder of this paper is organised as follows. Section 2 gives a brief overview of the ARM machine language and specialises a Hoare logic to reason about ARM machine code. Section 3 presents how the details of ARM instructions are captured by specifications in the new logic. Section 4 illustrates the use of the logic through examples. Section 5 presents the ARM model and Section 6 concludes with a summary.

2 Hoare Triples for ARM

This section instantiates a Hoare logic to ARM machine code. We start with a brief overview of ARM machine code and then describe how a general Hoare logic is specialised to reason about ARM code.

2.1 ARM Machine Code

ARM machine code runs on ARM processors. These are widely used commercial RISC processors often found in mobile phones. The resources that ARM instructions access are, from a birds-eye-view, the following:

1. 16 registers are visible at any time: register 15 is the program counter and the others are general purpose registers each holding a 32-bit value (by convention register 13 is the stack pointer and register 14 is the link register);
2. 4 status bits: negative, zero, carry and overflow;
3. a 32-bit addressable memory with entries of 8 bits (or equivalently, a 30-bit addressable memory with 32-bit entries).

This high-level view is sufficient for all 32-bit ARM instructions that do not require interaction between operation modes. In some sense these are the instructions of the "programmer's model" of ARM. The Hoare logic presented in this paper is restricted to the subset of 32-bit ARM instructions that can execute equally regardless of operation mode (user, supervisor, etc). However the operational model at the base of this work considers also the instructions that do depend on the operation mode, for more details see Section 5.

Some interesting features of the 32-bit ARM instructions that have required special attention are listed below.

1. All instructions can be executed conditionally, i.e. instructions can be configured to have no effect when the status bits fail to satisfy some condition.
2. All "data processing" instructions can update the status bits.
3. During execution, undefined instruction encodings or forbidden instruction arguments can be encountered, in which case the subsequent behaviour is implementation specific (modelled as *unpredictable* behaviour).

2.2 ARM Hoare Logic

This section specialises a general machine-code Hoare logic, presented earlier [5], to ARM machine code. The general logic specifies the behaviour of collections of code segments using Hoare triples that allow multiple entry points and multiple exit points. In this paper we will mainly use specifications with a single entry point and a single sequence of code:

$$\{P\}\ \ cs\ \ \{Q_1\}^{h_1} \cdots \{Q_k\}^{h_k}$$

Such specifications are to be read informally as follows: whenever P holds for the current state and code sequence cs is executed, a state will be reached where one of the postconditions Q_i holds and the program counter will have been updated by function h_i.

Models of states usually consist of tuples of components. However, when defining the semantics of our general Hoare logic, we have found it more convenient to represent states as sets of *basic state elements* that separately specify the values of single pieces of the state. This allows states to be split and partitioned using elementary set operations (e.g. $\cup$, $\cap$, $-$). The elements we need for ARM are: $\mathsf{Reg}\ i\ x$ (specifies register i has value x), $\mathsf{Mem}\ j\ y$ (specifies memory location j has value y), $\mathsf{Status}\ (s_n, s_z, s_c, s_v)$ (specifies the values of the four status flags), $\mathsf{Undef}\ b$ (specifies whether an 'undefined' instruction has been encountered) and $\mathsf{Rest}\ z$ (specifies the remainder of the state). Thus for ARM, each state will be a set of the form[1]:

$$\begin{array}{l}\{\ \mathsf{Reg}\ 0\ x_0\,,\ \mathsf{Reg}\ 1\ x_1\,,\ \mathsf{Reg}\ 2\ x_2\,,\ \cdots\,,\ \mathsf{Reg}\ 15\ x_{15}\,,\\ \ \ \mathsf{Mem}\ 0\ y_0\,,\ \mathsf{Mem}\ 1\ y_1\,,\ \mathsf{Mem}\ 2\ y_2\,,\ \cdots\,,\ \mathsf{Mem}\ (2^{30}{-}1)\ y_{(2^{30}-1)}\,,\\ \ \ \mathsf{Status}\ (s_n, s_z, s_c, s_v),\ \mathsf{Undef}\ b,\ \mathsf{Rest}\ z\ \}\end{array}$$

Fox's ARM model uses a tuple-like state representation, thus in order to specialise our general Hoare logic to his ARM model, we need a translation function from Fox's state representation to our set-based representation. Such a translation is defined as follows. Let $reg\ a\ s$ extract the value of register a from state s, $mem\ a\ s$ extract the value of memory location a from s and $status$ extract the value of the four status bits from s. Also let $s.undefined$ indicate whether s is considered as a state from which *unpredictable* behavior may occur

[1] Numerals denote both bit strings and natural numbers. Type annotations in the syntax of HOL4: $\mathsf{Reg}\ (i\texttt{:word4})\ (x\texttt{:word32})$ and $\mathsf{Mem}\ (j\texttt{:word30})\ (y\texttt{:word32})$.

and let *hidden* project the remaining part of an ARM state, i.e. the part that is not observable by *reg*, *mem*, *status* and $undefined$. We can then define:

$$\begin{aligned} arm2set(s) \;=\; & \{\ \mathsf{Reg}\ a\ (reg\ a\ s) \mid \text{any } a\ \} \cup \\ & \{\ \mathsf{Mem}\ a\ (mem\ a\ s) \mid \text{any } a\ \} \cup \\ & \{\ \mathsf{Status}\ (status\ s),\ \mathsf{Undef}\ s.undefined,\ \mathsf{Rest}\ (hidden\ s)\ \} \end{aligned}$$

The translation does not loose any information and therefore has an inverse $set2arm$ such that $\forall s.\ set2arm(arm2set(s)) = s$.

The general theory is formally specialised to reason about ARM machine code by instantiating a 6-tuple $(\Sigma, \alpha, \beta, next, pc, inst)$ that parametrises the general theory. Here Σ is the set of states, *next* is a next-state function $next : \Sigma \to \Sigma$, and $pc : \alpha \to \Sigma \to \mathbb{B}$ and $inst : \alpha \times \beta \to \Sigma \to \mathbb{B}$ are elementary assertions over states. The general theory is instantiated to the ARM model by setting Σ to be the range of $arm2set$, α to be the set of 30-bit addresses and β to be the set of 32-bit words. The next-state function is defined using the next-state function for the ARM model ($next\_arm$) and translations $arm2set$ and $set2arm$.

$$next(s) \;=\; arm2set(next\_arm(set2arm(s)))$$

In what follows *addr* is a function that transforms a 30-bit address to a 32-bit (word-aligned) address by appending two zeros as new least significant bits. The program-counter assertion $pc(p)$ is defined to check that a subset of a state implies that the program counter is set to p and that the state is well-defined. The instruction assertion $inst(p, c)$ makes sure that instruction c is stored in the location which is executed when the program counter has value p. These assertions are predicates on sets of basic state elements: $pc(p)$ is true of a set if it is $\{\ \mathsf{Reg}\ 15\ (addr(p)), \mathsf{Undef\ F}\ \}$ and $inst(p, c)$ is true of a set if it is $\{\ \mathsf{Mem}\ p\ x\ \}$. Thus:

$$\begin{aligned} pc(p) \;&=\; \lambda s.\ s = \{\ \mathsf{Reg}\ 15\ (addr(p)), \mathsf{Undef\ F}\ \} \\ inst(p, x) \;&=\; \lambda s.\ s = \{\ \mathsf{Mem}\ p\ x\ \} \end{aligned}$$

3 Instruction Specifications

The previous section discussed how we instantiate our abstract Hoare logic to ARM machine code. This section shows how the new Hoare triples capture the behaviour of basic ARM instructions. We start by explaining how a simple specification relates to the ARM model and then go on to show how the full generality of ARM instructions is captured by the new Hoare triples.

Consider the following specification of `SUB a,a,#1` (subtract by one).

$$\begin{array}{c} \{R\ a\ x\} \\ \texttt{SUB } a\texttt{,}a\texttt{,\#1} \\ \{R\ a\ (x{-}1)\}^{+1} \end{array}$$

This specification states that register a is decremented by one and that the program counter is incremented by one. Let $R\ r\ x = \lambda s.\ (s = \{\mathsf{Reg}\ r\ x\})$. In terms

of the set-based state representation the specification ought to be read as follows: whenever `SUB a,a,#1` is executed, the part of the state corresponding to $\{\mathsf{Reg}\ a\ x\}$ is updated to $\{\mathsf{Reg}\ a\ (x{-}1)\}$ and simultaneously the part corresponding to the program counter is updated by function $+1$ (abbreviates $\lambda n.\ n{+}1$), i.e. the subset corresponding to $\{\mathsf{Reg}\ 15\ (addr(p)), \mathsf{Undef\ F}\}$, for some value p, becomes $\{\mathsf{Reg}\ 15\ (addr(p{+}1)), \mathsf{Undef\ F}\}$.

In terms of the ARM model, the above specification is formally equivalent to the following. Let $run(k, s)$ be a function that applies $next\_arm\ k$ times to state s, and let $\lfloor\cdot\rfloor$ be a function that produces the 32-bit encoding of a given ARM instruction. Also let $\text{FRAME} = \{\ \mathsf{Reg}\ a\ x \mid \text{any } x\ \} \cup \{\ \mathsf{Reg}\ 15\ x \mid \text{any } x\ \}$.

$$
\begin{array}{l}
\forall s\ p.\ \ (reg\ a\ s = x) \wedge (reg\ 15\ s = addr(p)) \wedge (a \neq 15) \wedge \\
\qquad (mem\ p\ s = \lfloor \texttt{SUB a,a,\#1} \rfloor) \wedge \neg s.undefined \Rightarrow \\
\qquad \exists k.\ \ \text{let } s' = run(k, s) \text{ in} \\
\qquad\qquad (reg\ a\ s' = x{-}1) \wedge (reg\ 15\ s' = addr(p{+}1)) \wedge (a \neq 15) \wedge \\
\qquad\qquad (mem\ p\ s' = \lfloor \texttt{SUB a,a,\#1} \rfloor) \wedge \neg s'.undefined \wedge \\
\qquad\qquad (arm2set(s) - \text{FRAME} = arm2set(s') - \text{FRAME})
\end{array}
$$

For most part this expansion contains no surprises: whenever registers a is x, the program counter points at an encoding of `SUB a,a,#1` and the state is well-defined, then register a is decremented, the program counter is updated by function $+1$ and the state remains well-defined. The interesting part of the above specification is the last line. The last line states that the initial state is the same as the result state, if one removes registers a and 15 from both states. The last line specifies what is left unchanged, i.e. the scope of the operation.

The Hoare triples satisfy a frame rule for extending the scope. The frame rule is intuitively similar to that of separation logic [6]. The frame rule uses a separating conjunction ($*$). For $*$ define $split\ s\ (u, v)$ to mean that the pair of sets (u, v) partitions set s, i.e. $split\ s\ (u, v) = (u \cup v = s) \wedge (u \cap v = \emptyset)$, and then define $P * Q$ to be true if P and Q are true for disjoint parts of the state: $P * Q = \lambda s.\ \exists u\, v.\ split\ s\ (u, v) \wedge P\ u \wedge Q\ v$. The frame rule:

$$\frac{\{P\}\ c\ \{Q\}^h}{\forall F.\ \{P * F\}\ c\ \{Q * F\}^h}$$

The frame rule can be used to expand the basic specification of `SUB a,a,#1` to say that the value of register b stays constant, if b is distinct from a:

$$
\begin{array}{c}
\{R\ a\ x * R\ b\ y\} \\
\texttt{SUB a,a,\#1} \\
\{R\ a\ (x{-}1) * R\ b\ y\}^{+1}
\end{array}
$$

The expansion of the extended specification is equal to the above expansion with the inclusion of $(reg\ b\ s = y) \wedge (a \neq b) \wedge (b \neq 15)$ for both s and s'. The separating conjunction implies necessary inequalities as a result of its requirement of disjointness. We use $*$ as a basic building block in all our specifications.

The remainder of this section describes the generalisations that are made in order to accommodate the full features of real ARM instructions.

3.1 Conditional Execution

Every 32-bit ARM instruction can execute conditionally according to a condition code that is encoded in each instruction. The instruction is executed if the condition associated with the given condition code is satisfied by the status bits. If the condition is not satisfied then the instruction has no effect (other than incrementing the program counter). The behavior of conditional execution is captured by giving each instruction two specifications, one for the case when it has an effect and one for the case when it has no effect. Let $\textsc{pass}(c, z)$ assert that bits z satisfy condition code c. Let $\neg\textsc{pass}(c, z)$ be its negation. Let $S\ z = \lambda s.\ (s = \{\mathsf{Status}\ z\})$.

$$\begin{array}{cc} \{R\ a\ x * S\ z * \textsc{pass}(c,z)\} & \{S\ z * \neg\textsc{pass}(c,z)\} \\ \texttt{SUB}\,c\ \ a\texttt{,}a\texttt{,\#1} & \texttt{SUB}\,c\ \ a\texttt{,}a\texttt{,\#1} \\ \{R\ a\ (x{-}1) * S\ z\}^{+1} & \{S\ z\}^{+1} \end{array}$$

3.2 Status Bits

Most ARM instructions have a flag called the s-flag. When this flag is set, executing the command will update the status bits. Let $\mathit{sub\_status}(x, y)$ calculate the value of the four status bits for the subtraction $x{-}y$.

$$\begin{array}{c} \{R\ a\ x * S\ z * \textsc{pass}(c,z)\} \\ \texttt{SUB}\,c\,s\ \ a\texttt{,}a\texttt{,\#1} \\ \{R\ a\ (x{-}1) * S\ (\text{if } s \text{ then } \mathit{sub\_status}(x,1) \text{ else } z)\}^{+1} \end{array}$$

3.3 Addressing Modes

The `SUB` instruction, used above, can of course do more than subtract by one. It can subtract by any small (shifted/rotated) constant or a (shifted/rotated) register value. The form of the second term in a subtraction is specified by an *addressing mode* (for `SUB`: ARM Addressing Mode 1). Our specifications parametrise the addressing mode as a variable m. The functions $\mathit{encode\_am}_1$ and $\mathit{value\_am}_1$ construct, respectively, the instruction encoding and second argument of an arithmetic operation for a given instance m of ARM Addressing Mode 1. Examples:

$$\begin{array}{cc} \{R\ a\ x\} & \{R\ a\ x * R\ b\ y\} \\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,}\mathit{encode\_am}_1(m,a) & \texttt{SUB}\ \ b\texttt{,}b\texttt{,}\mathit{encode\_am}_1(m,a) \\ \{R\ b\ (x{-}\mathit{value\_am}_1(m,x))\}^{+1} & \{R\ a\ x * R\ b\ (y{-}\mathit{value\_am}_1(m,x))\}^{+1} \end{array}$$

Specifications, such as those shown below, can be produced, if we instantiate m appropriately and rewrite using the definitions of $\mathit{encode\_am}_1$ and $\mathit{value\_am}_1$.

$$\begin{array}{ccc} \{R\ a\ x\} & \{R\ a\ x * R\ b\ y\} & \{R\ a\ x * R\ b\ y\} \\ \texttt{SUB}\ \ a\texttt{,}a\texttt{,\#1} & \texttt{SUB}\ \ b\texttt{,}b\texttt{,}a & \texttt{SUB}\ \ b\texttt{,}b\texttt{,}a\texttt{,LSL\ \#5} \\ \{R\ a\ (x{-}1)\}^{+1} & \{R\ a\ x * R\ b\ (y{-}x)\}^{+1} & \{R\ a\ x * R\ b\ (y{-}(x{\ll}5))\}^{+1} \end{array}$$

3.4 Aligned Addresses

A 32-bit address is word aligned if it is divisible by four. On ARM, memory accesses to word-sized entities generally result in rotations of the accessed words, if the accessed address is not word aligned. In order to avoid cluttering specifications with details of word rotations, we specify word-aligned memory accesses separately from the general case. The specification for aligned load-word `LDR` requires no rotations. Let $R'\ r\ x$ assert that register r holds a word-aligned address x, i.e. $R'\ r\ x = R\ r\ (addr(x))$, and let $M\ a\ x = \lambda s.\ (s = \{\mathsf{Mem}\ a\ x\})$.

$$\begin{array}{c}\{R\ a\ z * R'\ b\ x * M\ (address\_am_2(m,x))\ y\}\\ \texttt{LDR}\ a\texttt{,}encode\_am_2(m,b)\\ \{R\ a\ y * R'\ b\ (writeback\_am_2(m,x)) * M\ (address\_am_2(m,x))\ y\}^{+1}\end{array}$$

The above can be specialised to the following by instantiation of m:

$$\begin{array}{c}\{R\ a\ z * R'\ b\ x * M\ x\ y\}\\ \texttt{LDR}\ a\texttt{,[}b\texttt{]}\\ \{R\ a\ y * R'\ b\ x * M\ x\ y\}^{+1}\end{array}\qquad\begin{array}{c}\{R\ a\ z * R'\ b\ x * M\ (x{-}1)\ y\}\\ \texttt{LDR}\ a\texttt{,[}b\texttt{,\#-4]!}\\ \{R\ a\ y * R'\ b\ (x{-}1) * M\ (x{-}1)\ y\}^{+1}\end{array}$$

3.5 Branch Instructions

Branch instructions are given one postcondition for each exit point. The specification of a conditional relative branch:

$$\begin{array}{c}\{S\ z\}\\ \texttt{B}c\ \texttt{\#}k\\ \{S\ z * \mathrm{PASS}(c,z)\}^{+(k+2)}\\ \{S\ z * \neg\mathrm{PASS}(c,z)\}^{+1}\end{array}$$

The intuition for multiple postconditions is that one of the postconditions will be reached. Whenever `B`c `#`k is executed, there will either be a jump of $k+2$ instructions or a jump to the next instruction. The formal semantics is based on disjunction, for details see our earlier paper [5].

3.6 Automation

The above specifications are rather hard to use in practice if addressing modes and condition codes have to be instantiated by hand. We found it useful to write an ML function that maps string representations of the instructions to their respective instantiations of the general specifications. The instantiating ML function was connected to an ML function that calculates the composition of a given list of instruction specifications using the composition rule from our earlier paper [5], e.g. the input `["LDR a,[b,#16]!","SUBS a,a,1","BNE k"]` gives:

$$\begin{array}{c}\{R\ a\ z * R'\ b\ x * M\ x\ y * S\ \_\}\\ \texttt{LDR}\ a\texttt{,[}b\texttt{,\#16]!; SUBS}\ a\texttt{,}a\texttt{,\#1; BNE \#}k\\ \{R\ a\ (y{-}1) * R'\ b\ (x{+}4) * M\ x\ y * S\ \_ * \langle y{-}1 \neq 0\rangle\}^{+(k+2)}\\ \{R\ a\ (y{-}1) * R'\ b\ (x{+}4) * M\ x\ y * S\ \_ * \langle y{-}1 = 0\rangle\}^{+3}\end{array}$$

Here the ML function treats x as a word-aligned address and hides the initial and final value of the status bits using an underscore (_) which denotes 'some-value' (formally _ is a postfix function: $P\ \_ = \lambda s.\ \exists x.\ P\ x$).

4 Case Studies

This section demonstrates how specifications from the previous section can be reformulated and combined in order to prove specifications for ARM code with loops, procedures and pointer-based data structures.

4.1 Factorial Program

As an initial example, we will show how loop rules can be proved and used. A loop rule will be proved for a count-down loop and then used in the proof of the following factorial program:

```
    MOV  b, #1     ; b := 1
L:  MUL  b, a, b   ; b := a × b
    SUBS a, a, #1  ; decrement a and update status bits
    BNE  L         ; if a is nonzero then jump to L
```

This program stores the factorial of register a (modulo 2^{32}) in register b, if a is initially non-zero. It calculates the factorial by executing a count-down loop:

```
b := 1; repeat { b := a × b; a := a - 1 } until (a=0)
```

Loop. A specification for a loop of the form "`L:` *body*; `SUBS` a`,`a`,#1; BNE L`" can be devised using the specification of the combined effect of `SUBS` and `BNE`. For the proof we will require that *body* has a specification of the following form. Let m be the length of the code sequence *body*.

$$\begin{array}{c} \{Inv(x) * R\ a\ x * S\ \_ * \langle x \neq 0 \rangle\} \\ body \\ \{Inv(x{-}1) * R\ a\ x * S\ \_\}^{+m} \end{array} \tag{1}$$

The technique described in Section 3.6 can be used to construct a specification for "`SUBS` a`,`a`,#1; BNE #`k", which can be composed with (1) to give:

$$\begin{array}{c} \{Inv(x) * R\ a\ x * S\ \_ * \langle x \neq 0 \rangle\} \\ body\texttt{; SUBS } a\texttt{,}a\texttt{,\#1; BNE \#}k \\ \{Inv(x{-}1) * R\ a\ (x{-}1) * S\ \_ * \langle x{-}1 \neq 0 \rangle\}^{+(m+k+3)} \\ \{Inv(x{-}1) * R\ a\ (x{-}1) * S\ \_ * \langle x{-}1 = 0 \rangle\}^{+(m+2)} \end{array}$$

A loop is constructed if k is assigned value $-(m{+}3)$, since the program counter update is then $+0$, i.e. the program counter returns to its original value. With a

few other simplifications we can reveal that the precondition is satisfied by each jump to the top of the loop. Let $<$ denote less-than over unsigned 32-bit words.

$$\begin{array}{c}\{Inv(x) * R\ a\ x * S\ \_ * \langle x \neq 0\rangle\} \\ body;\ \texttt{SUBS}\ a,a,\#1;\ \texttt{BNE}\ \#\text{-}(m{+}3) \\ \{\exists z.\ Inv(z) * R\ a\ z * S\ \_ * \langle z \neq 0\rangle * \langle z < x\rangle\}^{+0} \\ \{Inv(0) * R\ a\ 0 * S\ \_\}^{+(m+2)}\end{array}$$

Postconditions that describe a jump to a precondition, with some bounded variant that decreases at each jump, can be removed since the loops they describe will terminate and thus a different postconditions will eventually be reached [5]. The postcondition with update $+0$ is removed:

$$\begin{array}{c}\{Inv(x) * R\ a\ x * S\ \_ * \langle x \neq 0\rangle\} \\ body;\ \texttt{SUBS}\ a,a,\#1;\ \texttt{BNE}\ \#\text{-}(m{+}3) \\ \{Inv(0) * R\ a\ 0 * S\ \_\}^{+(m+2)}\end{array} \tag{2}$$

We have proved a loop rule: any code *body* and invariant Inv that satisfies specification (1) will also satisfy specification (2).

Factorial. The factorial program is easily proved in case we can find a specification of `MUL` that fits specification (1) from above. Notions of factorials and partial factorials are needed in order to create a suitable specification for `MUL`. Let fac be the factorial function over natural numbers:

$$fac(n) \quad = \quad \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n{-}1) & \text{if } n > 0 \end{cases}$$

Let factorial and partial factorial (e.g. $5 \times 4 \times 3 = fac(5)/fac(2)$) over 32-bit words be defined using conversion to and from the natural numbers, $w2n$: `word32->num` and $n2w$: `num->word32`.

$$\begin{array}{rcl} x! & = & n2w(fac(w2n(x))) \\ y \cdot\cdot\ x & = & n2w(fac(w2n(y))/fac(w2n(x))) \end{array}$$

Notable features of the partial factorial $(\cdot\cdot)$ are that $x \cdot\cdot\ 0 = x!$ and $y \cdot\cdot\ y = 1$ and $(z \cdot\cdot\ y) \times y = z \cdot\cdot\ (y{-}1)$, if $y \leq z$ and $y \neq 0$.

A specification for `MUL` can now be molded into the required form:

$$\begin{array}{c}\{R\ a\ x * R\ b\ (z \cdot\cdot\ x) * S\ \_ * \langle x \neq 0\rangle\} \\ \texttt{MUL}\ b,a,b \\ \{R\ a\ x * R\ b\ (z \cdot\cdot\ (x{-}1)) * S\ \_\}^{+1}\end{array}$$

The loop rule from the previous section then gives the following result:

$$\begin{array}{c}\{R\ a\ x * R\ b\ (z \cdot\cdot\ x) * S\ \_ * \langle x \neq 0\rangle\} \\ \texttt{MUL}\ b,a,b;\ \texttt{SUBS}\ a,a,\#1;\ \texttt{BNE}\ \#\text{-}4 \\ \{R\ a\ 0 * R\ b\ (z \cdot\cdot\ 0) * S\ \_\}^{+3}\end{array}$$

```
sum:   CMP    a,#0              ; compare a with 0
       MOVEQ  r15,r14           ; return, if a = 0
       STR    a,[r13,#-4]!      ; push a
       STR    r14,[r13,#-4]!    ; push link-register
       LDR    r14,[a]           ; temp := node value
       ADD    s,s,r14           ; s := s + temp
       LDR    a,[a,#4]          ; a := address of left
       BL     sum               ; s := s + sum of a
       LDR    a,[r13,#4]        ; a := original a
       LDR    a,[a,#8]          ; a := address of right
       BL     sum               ; s := s + sum of a
       LDR    r15,[r13],#8      ; pop two and return
```

Fig. 1. BINARY_SUM: ARM code to sum the values at the nodes of a binary tree.

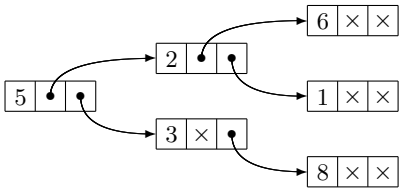
Instantiating z to x and composing a specification for MOV at the front yields a specification for the factorial program:

$$\{R\ a\ x * R\ b\ \_ * S\ \_ * \langle x \neq 0 \rangle\}$$
$$\texttt{MOV}\ b\texttt{,\#1; MUL}\ b,a,b\texttt{; SUBS}\ a\texttt{,}a\texttt{,\#1; BNE \#-4}$$
$$\{R\ a\ 0 * R\ b\ x! * S\ \_\}^{+4}$$

The final specification states that the program stores the factorial of register a (modulo 2^{32}) in register b, if the initial value of register a was non-zero.

4.2 Sum of Nodes in Binary Tree

Next we illustrate the proof of a recursive procedure that sums the values stored at the nodes of a binary tree. The implementation we prove is called BINARY_SUM. Its code is shown in Figure 1. BINARY_SUM makes a depth-first pass through a binary tree, where nodes are stored as blocks of three consecutive memory elements: one 32-bit value and two aligned addresses pointing to the root of the subtrees (called left and right). The procedure adds the sum of the tree with root at address a into register s. When executing BINARY_SUM on the tree depicted below, it adds the values 5, 2, 6, 1, 3, 8 to register s. The recursive calls are realised by the BL instruction.



Binary Tree. The trees `BINARY_SUM` traverses are modelled as trees that are either empty (Leaf) or a branch ($\mathsf{Node}(x, l, r)$). Each branch holds a 32-bit value x and two subtrees l and r. The sum of such a tree is defined as follows:

$$\begin{array}{rcl} sum(\mathsf{Leaf}) & = & 0 \\ sum(\mathsf{Node}(x,l,r)) & = & x + sum(l) + sum(r) \end{array}$$

A predicate $tree(x, t)$ is defined to assert that tree t is stored in memory with its root at address x. For ease of presentation we require that subtrees are stored in disjoint parts of the memory (which is implied by the occurrence of $*$ between the recursive assertions of $tree$). Here and throughout $M'\ a\ x$ asserts that memory location a holds aligned address x, i.e. $M'\ a\ x = M\ a\ (addr(x))$.

$$\begin{array}{rcl} tree(a, \mathsf{Leaf}) & = & \langle a = 0 \rangle \\ tree(a, \mathsf{Node}(x,l,r)) & = & \exists a_1\ a_2.\ M\ a\ x * M'\ (a{+}1)\ a_1 * M'\ (a{+}2)\ a_2 * \\ & & \quad tree(a_1, l) * tree(a_2, r) * \langle a \neq 0 \rangle \end{array}$$

The $tree$ assertion allows us to prove that "`LDR b,[a]; ADD s,s,b`" adds the value of a node, addressed by register a, to register s. Notice that the specification must mention register b, since the value of register b is updated by this operation.

$$\begin{array}{c} \{R'\ a\ x * R\ s\ z * tree(x, \mathsf{Node}(y,l,r)) * R\ b\ \_\} \\ \texttt{LDR}\ b\texttt{,[}a\texttt{]; ADD}\ s\texttt{,}s\texttt{,}b \\ \{R'\ a\ x * R\ s\ (z{+}y) * tree(x, \mathsf{Node}(y,l,r)) * R\ b\ \_\}^{+2} \end{array}$$

The above specification is a result of a composition of the specifications for `LDR` and `ADD`, an application of the frame rule, and a reformulation that introduces the existential quantifier hidden in $tree(x, \mathsf{Node}(y, l, r))$.

Stack. `BINARY_SUM` uses the stack to store local variables. In order to specify the stack operations, a notion of a stack segment is formalised. On ARM processors the stack is by convention descending, i.e. it grows towards lower addresses. The stack pointer, register 13, holds the address of the top element of the stack.

A stack predicate is defined using two auxiliary definitions: $ms(a, [x_0; \cdots; x_m])$ specifies that the 32-bit words $x_0, \cdots, x_n$ are stored in sequence from address a upwards in memory and $blank(a, n)$ asserts that n memory locations from address a downwards have 'some value'. The stack predicate $stack(sp, xs, n)$ is defined to assert that the aligned address sp is stored in register 13, that xs is the sequence of elements pushed onto the stack (above sp) and that there are n unused slots on top of the descending stack (immediately beneath sp).

$$\begin{array}{rcl} ms(a, [x_0; x_1; \cdots; x_m]) & = & M\ a\ x_0 * M\ (a{+}1)\ x_1 * \cdots * M\ (a{+}m)\ x_m \\ blank(a, n) & = & M\ a\ \_ * M\ (a{-}1)\ \_ * \cdots * M\ (a{-}(n{-}1))\ \_ \\ stack(sp, xs, n) & = & R'\ 13\ sp * ms(sp, xs) * blank(sp{-}1, n) \end{array}$$

The predicate $blank$ is needed in the above definition in order to state how much stack space is allowed to be used. As an example, consider the specification

for a stack push given below. The push instruction consumes one slot of stack space. Here cons is defined by $\mathsf{cons}\ x_0\ [x_1;\cdots;x_n] = [x_0;x_1;\cdots;x_n]$.

$$\begin{array}{c}\{R\ a\ x * stack(sp, xs, n{+}1)\}\\ \texttt{STR}\ a\texttt{,[r13,\#-4]!}\\ \{R\ a\ x * stack(sp{-}1, \mathsf{cons}\ x\ xs, n)\}^{+1}\end{array}$$

The verification of BINARY_SUM requires the pushed elements to be separated from the stack predicate at one point. The pushed elements can be extracted using the following equivalence. Let [] denote an empty list.

$$stack(sp, xs, n) \quad = \quad ms(sp, xs) * stack(sp, [], n)$$

Procedures. On ARM, procedures are by convention passed a return address in register 14 to which they must jump on exit. The control-flow contract of a procedure is enforced by a specification that requires the code to have a single exit point that updates the program counter to the address passed in register 14. If the program counter is initially p then the function $\lambda x.y$ updates the program counter to y, since $(\lambda x.y)\,p = y$.

$$\{P * R'\ 14\ y\}\ code\ \{Q * R\ 14\ \_\}^{\lambda x.y}$$

BINARY_SUM has the following procedure specification:

$$\begin{array}{l}\{R'\ a\ x * R\ b\ \_ * R\ s\ z * S\ \_ *\\ \quad tree(x,t) * stack(sp, [\,], 2 \times depth(t)) * R'\ 14\ y\}\\ \qquad\qquad \texttt{BINARY\_SUM}\\ \{R\ a\ \_ * R\ b\ \_ * R\ s\ (z + sum(t)) * S\ \_ *\\ \quad tree(x,t) * stack(sp, [\,], 2 \times depth(t)) * R\ 14\ \_\}^{\lambda x.y}\end{array}$$

Let $pre\ x\ t\ z\ y$ and $post\ x\ t\ z$ be the pre- and postcondition from above.

Procedure Calls and Recursion. The specification for BINARY_SUM is proved using induction. We induct on $depth(t)$ and assume that there is some code $\mathcal{C}$ that executes recursive calls correctly for any t' such that $depth(t') < depth(t)$.

$$\forall t'.\ depth(t') < depth(t)\ \Rightarrow\ \forall x\ z\ y.\ \{\,pre\ x\ t'\ z\ y\,\}\ \mathcal{C}\ \{\,post\ x\ t'\ z\,\}^{\lambda x.y}$$

With this assumption we can derive specifications for the BL instruction which perform the recursive calls in BINARY_SUM. The specifications are constructed using the proof rule derived in our earlier paper [5]. The code in these specifications is the union of the assumed code and the BL instruction:

$$\{\,pre\ x\ t'\ z\ \_\}\ \texttt{BL \#}k\ \cup\ \mathcal{C}\ \{\,post\ x\ t'\ z\,\}^{+1}$$

The rest of the verification is simple: compose the specifications for each instruction of BINARY_SUM in order to produce:

$$\{\,pre\ x\ t\ z\ y\,\}\ \texttt{BINARY\_SUM}\ \cup\ \mathcal{C}\ \{\,post\ x\ t\ z\,\}^{\lambda x.y}$$

An application of the following instance of complete induction over the natural numbers removes the imaginary code $\mathcal{C}$ and the assumption on t'.

$$\frac{\forall t\ \mathcal{C}.\ (\forall t'.\ depth(t') < depth(t) \Rightarrow \psi(t', \mathcal{C})) \Rightarrow \psi(t, code \cup \mathcal{C})}{\forall t.\ \psi(t, code)}$$

Tail-Recursion. `BINARY_SUM`, proved above, was constructed with clarity of presentation in mind. A good implementation would make use of the fact that the second recursive call can be made into a tail-recursive call. The last two instructions of `BINARY_SUM` are the following.

```
BL      sum             ; s := s + sum of a
LDR     r15,[r13],#8    ; pop two and return
```

These are turned tail-recursive by reversing the order as follows:

```
LDR     r14,[r13],#8    ; restore stack and link register
B       sum             ; s := s + sum of a
```

The new code copies the return address of the stack into the link register (register 14) rather than the program counter (register 15). It then performs a normal branch to the top of the procedure.

The optimised variant of `BINARY_SUM` is no harder to prove than the original version, normal composition is used instead of the rule for procedure calls. One can prove that the tail-recursive version requires only $2 \times ldepth(t)$ slots of stack space during execution. *ldepth* is defined as follows.

$$\begin{aligned} ldepth(\mathsf{Leaf}) &= 0 \\ ldepth(\mathsf{Node}(x, l, r)) &= \mathsf{max}(ldepth(l)+1, ldepth(r)) \end{aligned}$$

5 ARM Model

In Section 2.2, a Hoare logic for ARM machine code was constructed by placing a general Hoare logic on top of an operational model of the ARM instruction set. This section gives a brief overview of the ARM model that was used.

In the model underlying the Hoare triples, the state space is represented as a concrete HOL type (as opposed to a set of sets). The HOL type is a record type with four fields: *registers* (a mapping from register names to 32-bit words), *psrs* (a mapping from names of program status registers to 32-bit words), *memory* (a mapping from 30-bit words to 32-bit words) and *undefined* (a boolean indicating whether implementation specific behaviour follows from the current state).

The ARM Hoare triples only have access to 16 registers. However, the underlying model includes all 37 registers of an ARM processor. System modes have their own copies of some of the general purpose registers, thus the large number of register in total. The conceptual layout of the actual register bank

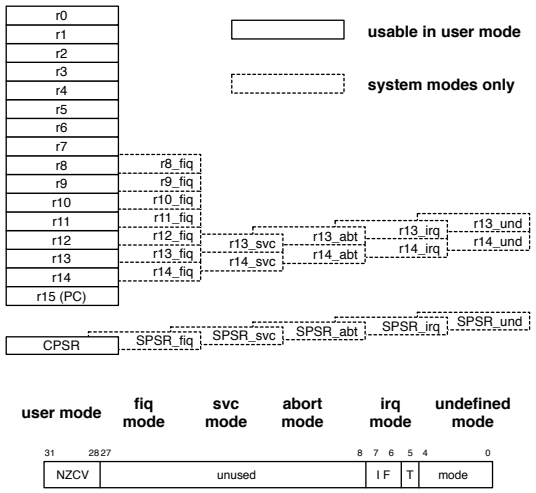
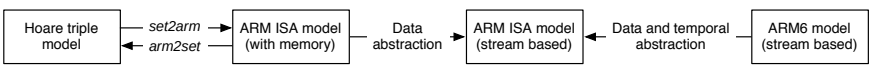


Fig. 2. ARM register banks and format of the Program Status Registers (PSRs).

is illustrated in Figure 2. The ARM Hoare triples convey the image of only 16 registers by presenting only the registers usable by the instructions of the current operation mode (for any mode, in case the Rest element is not mentioned in the precondition). This view of the registers is achieved by defining the functions *reg*, *status* and *hidden* (used in the definition of $arm2set$) to project the values of registers and status bits as viewed by the current operation mode, e.g. when operating in supervisor mode (svc), $reg\ 14\ s$ denotes the value of register r14_svc, $reg\ 2\ s$ is the value of register r2 and $reg\ 8\ s$ is the value of register r8_fiq.

The memory model deserves a comment, since a simple memory model is adopted: it is assumed that only data transfer instructions (memory stores) can alter the state of the memory i.e. the memory cannot be updated by the *environment*; when loading an instruction from memory, instruction pre-fetching (pipelining) is not considered; pre-fetch and data aborts are never raised i.e. it is assumed that one can always successfully access any memory address. Furthermore, input from the environment is not modelled i.e. it is assumed that there are no hardware interrupts. The Hoare logic that was instantiated in Section 2.2 can handle a more realistic model of memory, provided that it behaves as described above, for the part of memory mentioned in the precondition.

The ARM model used here is a conservative extension of a previously reported ARM model [2]. A well-understood path (by virtue of HOL theorems) exists between the ARM Hoare triples and a detailed register-transfer-level model of the hardware of an ARM processor. The path can be depicted as follows.



6 Summary

In this paper we have placed a general machine-code Hoare logic on top of a detailed model of the ARM machine language. By doing this we have constructed a framework that lifts reasoning from the tedious operational model to a manageable level. We have illustrated how specifications capture the generality of ARM instructions and demonstrated the use of the framework on examples that include loops, stacks, pointer data structures, procedures, procedural recursion and tail recursion. We have not yet applied the framework to large case studies, but we believe we have a methodology and implemented tools that will scale. Demonstrating this is the next phase of our research.

Acknowledgments. We would like to thank Joe Hurd, Konrad Slind and Thomas Tuerk for discussions and comments. The first author, Magnus Myreen, is funded by Osk.Huttusen Säätiö and EPSRC. The second author, Anthony Fox, is also funded by EPSRC.

References

1. Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.
2. Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*. Springer, 2003.
3. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher-order logic)*. Cambridge University Press, 1993.
4. David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In Panagiotis Manolios and Matthew Wilding, editors, *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2006.
5. Magnus O. Myreen and Michael J.C. Gordon. Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, LNCS, pages 568–582. Springer-Verlag, 2007.
6. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.
7. Gang Tan and Andrew W. Appel. A compositional logic for control flow. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.
8. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.