Master's Thesis

Radboud University Nijmegen

Asm3: modeling assembly languages efficiently in Why3

Author:
Jonathan Moerman

Supervisor:
Freek Wiedijk

Second reader:
Marc Schoolderman

July 15, 2021

Abstract

In this thesis, I present a domain-specific language for modeling straight-line assembly code, along with a plugin that extends Why3 with support for this language. This plugin contains its own VC generator that emits verification conditions that generally can be proven in less time and by more provers compared to what Why3 would generate for equivalent WhyML code. This VC generator differs among other things, in that, for each sub-goal, it automatically omits premises that do not appear useful. The language has an alternative to Why3's `ref` references, which, unlike those references, supports aliasing and offsets while not increasing the verification time.

This thesis starts with a short introduction to Why3 and a list of shortcomings of using Why3's WhyML language for modeling assembly code. I introduce the Asm3 language and the VC generator of the Asm3 plugin. I demonstrate how this plugin performs compared to plain Why3, using a short 16×16 multiplication AVR assembly routine as an example. Finally, I show what the effects of the Asm3 plugin's VC generation optimizations are on the verification times and the number of goals each prover was able to verify, using a 64×64 Karatsuba multiplication routine as a benchmark.

Contents

1 Introduction

Software verification is an important process with which we can help make sure that mission-critical code doesn't crash or fail and make sure that cryptographical software doesn't leak secrets. Unfortunately, this is often a complex, tedious, and time-intensive process.

Verifying properties of highly optimized assembly code can be especially tricky as routines can be hundreds to thousands of instructions long, contain steps that are interleaved with other steps, and contain tricks that are non-trivial to reason about. The number of instructions is not only larger than the number of statements in high–level programming languages, each instruction may have a large number of effects. For example, an instruction may update a register but also update 5 flags. In many cases, only a few of these effects are relevant for the result of a piece of code. For example, the next instruction may already overwrite all status flags set by the previous instruction, without using any of these flags to generate its output.

Marc Schoolderman and I have previously used the Why3 platform to formally verify a high-speed implementation of the Curve25519 key agreement scheme for 8-bit AVR microcontrollers [13]. While modeling the AVR instruction set in Why3's WhyML language and translating the assembly code to WhyML using this model was generally straightforward, we hit several issues.

Among other things, the verification conditions (See page 10) Why3 discharged contained a lot of hypotheses that weren't useful for the current proof goal. Another issue was that due to Why3's strict aliasing requirements it wasn't feasible to model registers with separate variables. This made it more complicated to specify which registers were modified by a section of code and also negatively impacted the verification of the generated goals. When comparing these verification conditions with hand-made verification conditions that did not suffer from these issues, the verification times of Why3's verification conditions generally were higher and fewer provers were able to verify these verification conditions.

In this thesis, I will present a proof-of-concept plugin that allows for the generation of verification conditions that are generally both easier to read by users, and easier to prove for automated theorem provers. This plugin provides a way to automatically filter out hypotheses that are unlikely to positively contribute to the proof of the current goal. It also provides a means to model the value stored in objects such as registers separately while allowing aliasing and working with offsets of these objects.

2 Why3

Why3 [8] is a platform for deductive program verification. This platform features a language, WhyML, that is suitable for modeling the behavior of most programs while also allowing the user to formally specify the semantics of the code using annotations. Why3 automates the process of proving these properties as much as possible. Why3 generates proof obligations for the annotated programs and can encode these problems into the input of a wide range of automated theorem provers and interactive proof assistants. In many cases, SMT solvers such as *Z3*, *CVC4*, or *Alt-Ergo* or ATP systems such as *E* or *Vampire* can automatically verify that a property holds, while proof assistants (*Isabelle/HOL*, *Coq,* or *PVS*) can be used in cases where automated theorem provers fail.

2.1 WhyML

Why3 provides an ML-like language called WhyML. WhyML can be divided into a logical specification language part [2] and an ML-like programming language part. Similarly, the world of Why3 can be divided into a logic and a program part. The logic part contains mathematical constructs such as predicates, lemmas, and mathematical functions while the program part contains variables and program functions. The

`function f (x: a) : b = X` defines a logic function `f` of type `b` taking an argument of type `a`. The definition of `f` is `X`.

`let f (x: a) : b S = X` defines a program function `f` of type `b` taking an argument of type `a`. `S` is the specification given for `f` while `X` is the body of this function.

`val f (x: a) : b S` axiomatically defines a program function `f` of type `b` taking an argument of type `a`. Axiomatic here means that no implementation of this `f` is given. The specification `S` is used as an axiom to describe the behavior of `f`.

`m[x]` this returns element of the map `m` stored at key `x`. `[]` is the map-get operator.

`m[x <- y]` this returns a copy of `m` with the value stored at `x` replaced with `y`. `[<-]` is the map-set operator.

Table 1: Important WhyML syntax descriptions

programming part of WhyML isn't constrained to just specifying program functions: lemmas and mathematical functions can be constructed with it as well, as long as the code is purely functional[1]. This can be useful as, for example, this allows defining a mathematical function and a program function at the same time. Another advantage is that it is sometimes easy to provide an algorithm to prove a lemma. An example of some WhyML code is given in listing 1. Table 1 describes the WhyML syntax essential to understanding the listings contained in this thesis.

Listing 1: Example of some simple definitions in WhyML

```
module ExampleModule

  (* Use module Int from standard library file int *)
  use int.Int

  use ref.Ref

  (* functions can be defined in WhyML's logic language *)
  function xor (x y: bool) : bool = (x \/ y) /\ not (x /\ y)

  lemma double_xor_ident: forall x y. xor (xor x y) y = x

  (* functions can be logic and program functions *)
  (* fib is both a logic and a program function *)
  let rec function fib (n: int) : int
    requires { n >= 0 }
    (* fib terminates because n always decreases
       while remaining >= 0 *)
    variant  { n }
  = if n = 0 then 0 else
    if n = 1 then 1 else
    fib (n-1) + fib (n-2)

  (* iterative_fib is just a program function *)
  let iterative_fib (n: int) : int
    requires { n >= 0 }
    ensures { result = fib n }
  = if n = 0 then 0 else
    let prev = ref 0 in
    let current = ref 1 in
    for i = 2 to n do
```

[1]The code doesn't modify or depend on any global state.

```
      invariant { !current = fib (i-1) }
      invariant { !prev = fib (i-2) }
      let new_prev = !current in
      current := !current + !prev;
      prev := new_prev
    done;
    !current

  (* Lemmas can be defined using code.
     This is equivalent to:
  lemma fib_is_pos : forall n:int. n >= 0 -> fib n >= 0
     But here we already provide the induction steps. *)
  let rec lemma fib_is_pos (n: int)
    requires { n >= 0 }
    ensures { fib n >= 0 }
    variant { n }
  = if n > 1 then begin
      fib_is_pos (n-2);
      fib_is_pos (n-1)
    end

end
```

An example of how working with Why3's IDE looks like is shown in figures 1 and 2. These figures give a glimpse into the process of writing WhyML code proving specifications of this code in Why3's IDE.

2.2 Proving program properties

The formal specification of WhyML programs is provided by specifying preconditions (for what initial state is the behavior defined) and postconditions (what are the effects of the program; what does it do). Preconditions are given by `requires { ··· }` annotations while postconditions are given by `ensures { ··· }` annotations. Why3 will create a proof obligation each time a program function with preconditions is called to check that these preconditions are met. It uses the postconditions of each called function to encode the resulting state after executing this function. For each program function, Why3 will generate a single logic formula, a verification condition, that holds if and only if, given all preconditions are satisfied, the program function executes correctly and terminates in a state where all postconditions hold.

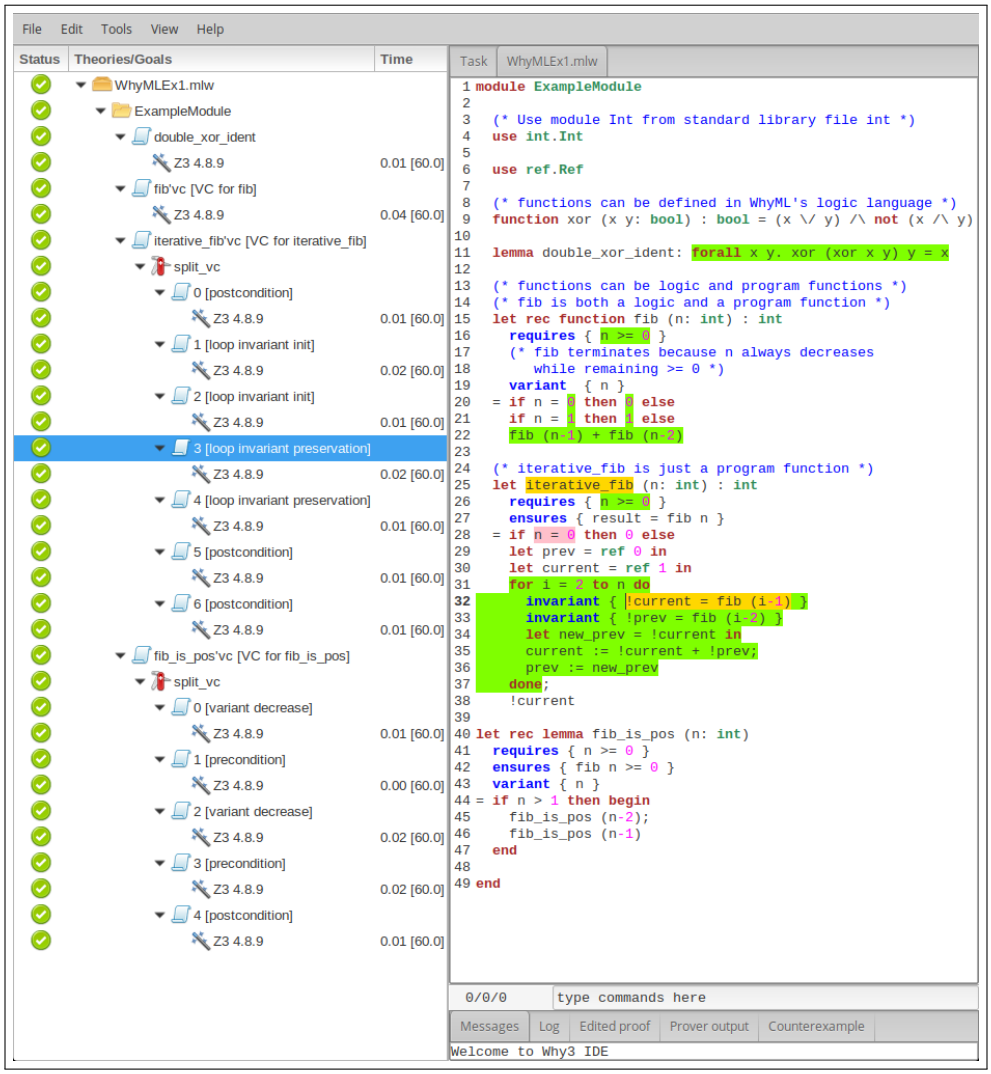


Figure 1: Interface of the Why3 IDE: Working with the WhyML code from listing 1

The right pane shows the WhyML code in the IDE's editing tab, while the left pane shows the proof goals generated for this code. Currently a sub-goal of the proof goal generated for `iterative_fib` is selected. The IDE highlights the source of the sub-goal with yellow, while sections of code that are used to generate this goal are highlighted with green. Formulas of which the negation is used are highlighted red.

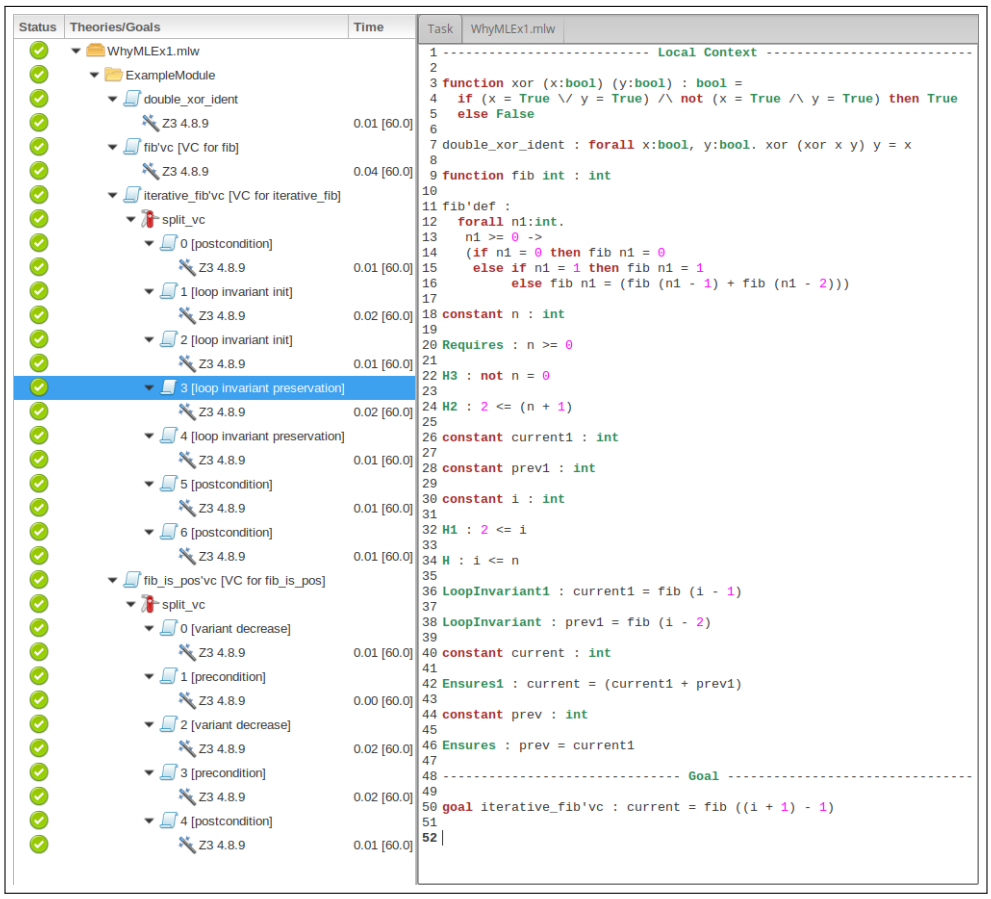


Figure 2: Interface of the Why3 IDE: Showing a sub-goal generated from the WhyML code in listing 1

2.2.1 Defining bytes in WhyML

To look at how we can prove the specification of a program function we will generate a verification condition for a simple function operating on bytes. If we want to model code working with bytes in WhyML, we would first need to model what a byte is. Listing 2 provides an example of a type `byte` along with a corresponding definition of addition. The infix function `+` (`let (+) ⋯` in listing 2) defines addition in terms of integer addition (addition in $\mathbb{Z}$). This program function has `ensures { byte'int result = byte'int b1 + byte'int b2 }` as postcondition, which states that the integer projection of the output[2] is equal to the sum of the integer projections of its input. This can only hold if the sum of the inputs would fit in a byte. This requirement is encoded in the precondition `requires { in_bounds (byte'int b1 + byte'int b2) }`.

Listing 2: Modeling bytes in Why3's WhyML language

```
type byte = < range 0 255 >

(*
Currently very little is defined for type byte.
We just have the projection function byte'int which gives the
projection of a byte in ℤ (Why3 type int).
We can use this to define operations on bytes.
*)

predicate in_bounds (n: int) = 0 <= n <= 255

(*
byte'int is only a logic function and not a program function,
so we cannot call it from a program function.
We can use it in the specification of a program function,
though.
*)
val to_int (n: byte) : int
  ensures { result = byte'int n }

val of_int (n: int) : byte
  requires is_in_bounds { in_bounds n }
  ensures { byte'int result = n }

(* Defining addition for bytes *)
let (+) (b1 b2: byte) : byte
  requires { in_bounds (byte'int b1 + byte'int b2) }
  ensures { byte'int result = byte'int b1 + byte'int b2 }
= return of_int (to_int b1 + to_int b2)
```

We can let Why3 automatically insert `byte'int` when it encounters a `byte` where it expects an `int`. This can be accomplished by adding the following line below the definition of `byte`:

```
meta coercion function byte'int
```

As leaving out all applications of `byte'int` makes the specifications significantly easier to read we will no longer explicitly use it past this point.

2.2.2 Building a simple verification condition for `double`

Now we have this basic definition we can write some simple program functions operating on bytes and verify that their specification holds. Listing 3 contains a program

[2] The keyword `result` denotes the output of a program function.

function `double` which returns a value that is twice as large as its input.

Listing 3: WhyML program function double

```
let double (b: byte): byte
  requires result_fits { in_bounds (2 * b) }
  ensures { result = 2 * b }
= b + b
```

We will now show how the specification of a program function such as `double` is proven in Why3. To verify that the specifications of programs are correct, Why3 generates Verification Conditions (VCs). VCs are logical formulas that are generated to help prove that the specifications of code are correct. A VC implies that given that the preconditions of a piece of code hold, the specification of its body is consistent and after execution the postconditions hold.

Why3 generates VCs using a standard weakest-precondition procedure [8]. The weakest precondition [6] of a statement S with respect to a formula Q, is a formula encoding the most general initial state such that, when executing S

- Execution does not go wrong. (i.e. assertions do not fail and the preconditions of program functions called in S are met.)
- Upon termination Q holds.

This weakest precondition of a statement S with respect to a formula Q is given by $wp(S,\ Q)$. For straight-line WhyML code we can approximate the weakest precondition of a statement with the following two rules:

- $wp(S,\ Q) \approx pre_S \wedge (post_S \rightarrow Q)$: the preconditions pre_S of statement S hold and the postconditions $post_S$ imply that Q holds.
- $wp(S_1\,;S_2,\ Q) = wp(S_1,\ wp(S_2,\ Q))$ gives the weakest precondition of the sequence $S_1\,;S_2$.

These rules only approximate the weakest precondition as they do not state how program variables are modeled. Section 2.2.3 provides some information on this topic.

We can apply these rules to generate a verification condition for `double`. For this program function we need to prove that the postcondition `result = 2* b` holds given `in_bounds (2* b)`. We haven't defined how program variables such as `b` and `result` are used to build the weakest precondition. For now, we will replace these variables $[\![\texttt{b}]\!]$ and $[\![\texttt{result}]\!]$ which should be interpreted as representing the semantics of `b` and `result` respectively. As the body of `double` consists of the single statement `b + b` we need to generate $\mathit{in\_bounds}(2 \times [\![\texttt{b}]\!]) \rightarrow wp(\texttt{b + b},\ [\![\texttt{result}]\!] = 2 \times [\![\texttt{b}]\!])$. The statement `b + b` has `in_bounds (b + b)` as precondition and `result = b + b` as postcondition, meaning that we can approximate the weakest precondition of `double` with $\mathit{in\_bounds}(2 \times [\![\texttt{b}]\!]) \rightarrow \mathit{in\_bounds}([\![\texttt{b}]\!] + [\![\texttt{b}]\!]) \wedge ([\![\texttt{result}]\!] = [\![\texttt{b}]\!] + [\![\texttt{b}]\!] \rightarrow [\![\texttt{result}]\!] = 2 \times [\![\texttt{b}]\!])$

Figure 3 shows the proof obligation Why3 generates for `double`. Observe that this closely matches the formula we generated previously. While almost any automatic theorem prover will be able to verify the goal formula of this VC, often it is necessary to verify a sub-goal (such as an assertion) individually. After introducing the bound variables and splitting the formula at the conjunction we are left with the more readable sub-goals shown in figures 4 and 5.

```
1 -------------------------- Local Context --------------------------
2
3 type byte = <range 0 255>
4
5 function byte'int byte : int
6
7 constant byte'maxInt : int = 255
8
9 constant byte'minInt : int = 0
10
11 predicate in_bounds (n:int) = 0 <= n /\ n <= 255
12
13 ------------------------------ Goal ------------------------------
14
15 goal double'vc :
16   forall b:byte.
17    in_bounds (2 * b) ->
18    in_bounds (b + b) /\
19    (forall result:byte. result = (b + b) -> result = (2 * b))
```

Figure 3: Verification condition containing all proof obligations generated for `double`. This image is a crop of a screenshot of the Why3 IDE window.

```
1 -------------------------- Local Context --------------------------
2
3 type byte = <range 0 255>
4
5 function byte'int byte : int
6
7 constant byte'maxInt : int = 255
8
9 constant byte'minInt : int = 0
10
11 predicate in_bounds (n:int) = 0 <= n /\ n <= 255
12
13 constant b : byte
14
15 result_fits : in_bounds (2 * b)
16
17 ------------------------------ Goal ------------------------------
18
19 goal double'vc : in_bounds (b + b)
```

Figure 4: First sub-goal of the VC of `double`: the precondition of `b + b`

```
1 --------------------------- Local Context ---------------------------
2
3 type byte = <range 0 255>
4
5 function byte'int byte : int
6
7 constant byte'maxInt : int = 255
8
9 constant byte'minInt : int = 0
10
11 predicate in_bounds (n:int) = 0 <= n /\ n <= 255
12
13 constant b : byte
14
15 result_fits : in_bounds (2 * b)
16
17 constant result : byte
18
19 Ensures : result = (b + b)
20
21 ------------------------------- Goal -------------------------------
22
23 goal double'vc : result = (2 * b)
```

Figure 5: Second sub-goal of the VC of `double`: the postcondition of `double`

2.2.3 Mutable variables

So far, we have only shown code without mutable variables. WhyML is also suitable for modeling imperative code. We can, for example, easily model mutable variables by using Why3's `ref` references. Why3 includes the `writes` annotation to specify which mutable variables will be modified and the `reads` annotation to specify whether the values initially stored in mutable variables are used by the code. In Why3 version 1.3, these annotations are often optional: Why3 will derive this information itself. `quadruple_by_ref` in listing 4 gives an example of a program function working with mutable variables. It updates the value where `b` points to. Here `:=` denotes assignment and the `!` operator returns the value to which a reference points.

While the programming part of Why3 supports mutable variables, the logic part does not. Why3 solves this issue by modeling mutable variables by creating a new incarnation of a variable each time a program this variable is updated. For example the sequence `x := 0; x := 1; x := 2; assert {!x > 1}` can be encoded as $\forall_{x_1,x_2,x_3 \in \mathbb{Z}}\, x_1 = 0 \land x_2 = 1 \land x_3 = 2 \rightarrow x_3 > 1$ where x_1 represents the value stored in `x` after executing `x := 0`, x_2 represents the value after executing `x := 1`, and x_3 represents the value after executing `x := 2`.

Listing 4: WhyML program function that quadruples a value passed by reference

```
use ref.Ref

let quadruple_by_ref (b: ref byte) : unit
  requires result_fits { in_bounds (4 * !b) }
  ensures { !b = 4 * old !b }
  reads { b }
  writes { b }
= b := !b + !b;
  b := !b + !b
```

The user can refer to previous values stored in a mutable variable or field by using the `at` and `old` keywords. The keyword `old` allows us to refer to the state prior to execution, while the keyword `at` can be used to refer to the state at a specific point during execution. When using `at`, a label is used to specify the specific state during execution. This label doesn't affect the execution of the code, but simply allows us to assign a name to a specific moment during execution. Listing 5 provides a simple demonstration of how `at` can be used to refer to a previous state.

Listing 5: WhyML fragment showing how the keyword `at` can be used.

```
...
x := 0;
label L in y = ···;
...
x := !x + !y;
assert { !x = !y by ((!x = 0) at L) }
(* This is equivalent to  (!x = 0) at L /\ (!x = 0) at L -> !x = !y *)
```

Using the information from above, we can improve our previous approximation of the weakest precondition of straight-line WhyML code:

- $wp(S,\ Q) = pre_S \land (\forall_{i_0} \cdots \forall_{i_n}\ post_S \rightarrow Q')$ where i_0 through i_n are new incarnations of the program variables v_0 through v_n modified by S. Q' is Q with v_0 through v_n replaced with these new incarnations, **unless** this variable appears under either `at` or `old`. For example: $wp(\llbracket \texttt{a:=!a+1} \rrbracket\ ,\ \llbracket \texttt{!a} \rrbracket_{\text{ub}} > \llbracket \texttt{old !a} \rrbracket_{\text{ub}})$ (where $\llbracket \texttt{!a} \rrbracket_{\text{ub}}$ is an unbound variable modeling `!a`): results in $\forall a_{new}\ a_{new} =$

$[\![\texttt{!a}]\!]_{\text{ub}} + 1 \rightarrow a_{new} > [\![\texttt{old !a}]\!]_{\text{ub}}$ given $post_S$ is defined as $a_{new} = [\![\texttt{!a}]\!]_{\text{ub}} + 1$ and pre_S is defined as *true*.

- $wp(\texttt{label L in } S,\ Q) = remove\_label(\texttt{L}, wp(S,\ Q))$. Where *remove_label* removes the given label from its second argument.
  If $wp(S,\ Q) = Q'$ and $remove\_label(\texttt{L}, Q') = Q''$ then Q'' is Q' with every sub-formula "`X at L`" is replaced with "`X`" in Q'. Unbound program variables appearing in `X` are now available to be replaced by incarnations generated by subsequent applications of the first rule.

- $wp(S_1;S_2,\ Q) = wp(S_1,\ wp(S_2,\ Q))$, this rule remains unchanged.

We can now apply these rules to generate the weakest precondition for `quadruple_by_ref` (here $[\![\texttt{!b}]\!]_{\text{ub}}$ is an unbound variable modeling `!b`):

$$
\begin{aligned}
wp(\texttt{b:=!b+!b;b:=!b+!b},\ [\![\texttt{!b}]\!]_{\text{ub}} = 4 \times [\![\texttt{old !b}]\!]_{\text{ub}}) &= \\
wp(\texttt{b:=!b+!b},\ wp(\texttt{b:=!b+!b},\ [\![\texttt{!b}]\!]_{\text{ub}} = 4 \times [\![\texttt{old !b}]\!]_{\text{ub}})) &= \\
wp(\texttt{b:=!b+!b},\ \mathit{in\_bounds}([\![\texttt{!b}]\!]_{\text{ub}} + [\![\texttt{!b}]\!]_{\text{ub}}) \wedge & \\
(\forall_{b_0}\, b_0 = [\![\texttt{!b}]\!]_{\text{ub}} + [\![\texttt{!b}]\!]_{\text{ub}} \rightarrow b_0 = 4 \times [\![\texttt{old !b}]\!]_{\text{ub}})) &= \\
\mathit{in\_bounds}([\![\texttt{!b}]\!]_{\text{ub}} + [\![\texttt{!b}]\!]_{\text{ub}}) \wedge (\forall_{b_1}\, b_1 = [\![\texttt{!b}]\!]_{\text{ub}} + [\![\texttt{!b}]\!]_{\text{ub}} \rightarrow & \\
\mathit{in\_bounds}(b_1 + b_1) \wedge (\forall_{b_0}\, b_0 = b_1 + b_1 \rightarrow b_0 = 4 \times [\![\texttt{old !b}]\!]_{\text{ub}})) &
\end{aligned}
$$

We now need to bind the remaining unbound program variables and add the precondition:

$$
\begin{aligned}
\forall_{b_2}\, \mathit{in\_bounds}(4 \times b_2) \rightarrow \mathit{in\_bounds}(b_2 + b_2) \wedge & \\
(\forall_{b_1}\, b_1 = b_2 + b_2 \rightarrow \mathit{in\_bounds}(b_1 + b_1) \wedge (\forall_{b_0}\, b_0 = b_1 + b_1 \rightarrow b_0 = 4 \times b_2)) &
\end{aligned}
$$

When we look at the VC generated by Why3 itself, shown in figure 6, we can see that this VC is equivalent to the formula we just generated.

```
1 --------------------------- Local Context ---------------------------
2
3 type byte = <range 0 255>
4
5 function byte'int byte : int
6
7 constant byte'maxInt : int = 255
8
9 constant byte'minInt : int = 0
10
11 predicate in_bounds (n:int) = 0 <= n /\ n <= 255
12
13 ------------------------------ Goal ------------------------------
14
15 goal quadruple_by_ref'vc :
16   forall b:byte.
17    in_bounds (4 * b) ->
18    in_bounds (b + b) /\
19    (forall o:byte.
20      o = (b + b) ->
21      (forall b1:byte.
22        b1 = o ->
23        in_bounds (b1 + b1) /\
24        (forall o1:byte.
25          o1 = (b1 + b1) -> (forall b2:byte. b2 = o1 -> b2 = (4 * b)))))
26
```

Figure 6: The VC generated for `quadruple_by_ref` by Why3

3 Modeling assembly code in Why3

This chapter describes some of the factors that make analyzing and verifying assembly code difficult. Section 3.2 describes techniques that have previously been used to combat these issues. Section 3.3 describes difficulties that remain even when these techniques are used.

3.1 What makes verifying assembly code difficult?

Assembly code has some properties that make it more difficult to analyze and verify than most code written in high level languages. In this thesis I will focus on two aspects that complicate the verification of assembly code: the prevalent occurence of aliasing and the length of assembly routines.

3.1.1 Aliasing

One issue when verifying assembly code is that aliasing occurs often. To see why this would be an issue consider the program function `add` in listing 6. Why3 requires that aliasing of mutable arguments must be statically known and will thus not allow `rd` and `rr` to alias. One could annotate `add` with `alias {rd with rr}`, but in that case, `rd` and `rr` **must** alias.

Listing 6: WhyML program function which adds the value stored in two 8 bit registers and stores the lowest 8 bits of the result in the first register and the carry bit in flag cf.

```
type reg = ref byte (* registers *)

val cf: ref bool (* carry flag *)

let add (rd rr: reg) : unit
  ensures {
    let result = old (!rd + !rr) in
    !rd = mod result 256 /\
    !cf = not div result 256 = 0
  }
  writes { rd, cf }
= let result = to_int !rd + to_int !rr in
  rd := of_int (mod result 256);
  cf := not div result 256 = 0
```

It is, of course, possible to model functions in which arguments may alias. We could, for example, model such arguments as indices into a single mutable map[3]. `add` could then be modeled as shown in listing 7.

Aside from solving the aliasing issue this also allows us to work with offsets, which is desirable when working with registers and essential when modeling main memory. Otherwise, we would need to pass a reference for every byte that is read from or written to, which would not only be cumbersome but in many cases impossible[4].

Modeling mutable objects such as registers using indices into a single mutable object has its disadvantages. All aliasing is allowed by default, meaning that the user must write preconditions to explicitly exclude all undesired aliasing cases. As long as the indices used are constants, the verification conditions Why3 generates are generally simple enough that most provers perform well. However, providing indices as arguments to the program function complicates these verification conditions to such a degree that provers perform much worse. In practice, this means that it takes more effort to verify functions that generalize a recurring step in an assembly routine for which the registers used vary. Another disadvantage is that tracking which values were

[3] In Why3's standard library the type map is defined as `type map 'a 'b = 'a -> 'b`

[4] Example: a function that copies n bytes of memory, where n is passed as an argument.

Listing 7: Alternative implementation of `add`

```
val regs : ref (map int byte) (* registers *)
val cf: ref bool (* carry flag *)

val reg_set (r : int) (b : byte) : unit
  ensures { !regs = !(old regs)[r <- b] }
  writes { regs }

let add (rd rr: int) : unit
  ensures {
    let result = !(old regs)[rd] + !(old regs)[rr] in
    (*implicit byte'int*) !regs[rd] = mod result 256 /\
    !cf = not div result 256 = 0
  }
  ensures {
    !regs = !(old regs)[rd <- !regs[rd]]
  }
  writes { regs, cf }
= let result = to_int !regs[rd] + to_int !regs[rr] in
  reg_set rd (of_int (mod result 256));
  cf := not div result 256 = 0
```

modified becomes more expensive, both in the time needed to write specifications and verification times of these specifications. Why3 can automatically track what memory has been modified and what remained unchanged if simple `ref` objects are used to model memory. This is not the case when modeling a memory address space in a single object. In that case, the user will need to manually specify all writes in the `ensures` part of the specification. These postconditions result in extra premises for any subsequent goals, which has a cost.

3.1.2 Code length

Highly optimized assembly code often consists of a large number of instructions. Branching is often kept to a minimum as jumping is costly[5] and having separate branches may leak information on the internal state (for example: through difference in timing or energy usage). The latter issue is of great concern if the code is handling sensitive data. Steps may also be interleaved to best make use of resources such as the available registers. For example, in one step some values may be produced which are needed for a subsequent step. It may be the case that, not enough registers are available to efficiently complete the current step while keeping these values untouched. Here, it could be better to execute a part of this next step and write the results to main memory. If these steps are performed sequentially then these values would have to be temporarily stored in main memory and retrieved in the subsequent step. Meanwhile, the results may still need to be stored in main memory. In this case, executing the steps sequentially would result in an extra store and load instructions having to be executed.

The minimal amount of branching combined with interleaved steps, and possibly other optimizations to best utilize the still available resources, can result in large intricate pieces of code that cannot neatly be divided into separate parts. This division would enable, or at least simplify, the separate analysis of the steps present in an assembly routine. It would allow for straightforward proof reuse if a continuous section of code reoccurs multiple times.

While instructions perform simple operations they can have a large number of

[5] Jumping itself adds a bit of overhead, but the main issue is that, if the branch predictor doesn't predict the right branch, the instruction pipeline has to be flushed.

effects[6], though this is highly dependent on the instruction set architecture (ISA). Take addition, in a language such as C the statement `a = a + b` will simply add the values of `a` and `b` together and store the result in `a`. In AVR assembly, the instruction `ADD R2, R3` not only adds the values stored in registers `R2` and `R3` together and stores the result in `R2`, but it will also update a number of flags. Specifically, it will update

- the carry flag (did the addition cause an overflow?)
- the zero flag (is the new value stored in `R2` all zeroes?)
- the negative flag (is the sign bit of `R2` now 1?)
- the two's complement overflow indicator (does the most significant bit of the result differ from the inputs?)
- the sign flag
- the half carry flag (did the addition of the lowest 4 bits result in a carry?)

While some of these effects may influence the end result of the code it is unlikely that all of these effects have an impact.

The large number of instructions combined with potentially a large number of effects per instruction results in a large number of premises in the generated VCs. This can be an issue as provers tend to perform worse as the number of premises increases.

3.2 Previous techniques used when verifying assembly code with Why3

3.2.1 Abstract blocks

One intuitive step to deal with the amount of premises is to break the code up into smaller chunks. In WhyML it is possible to divide code into *abstract blocks*. Abstract blocks hide their contents from any subsequent code. Only the specification given by the user and the information on what memory is modified are exported. These abstract blocks allow us to summarize the relevant effects while discarding all irrelevant information. If necessary, abstract blocks can be nested. The advantage of using abstract blocks instead of dividing the code into multiple program functions is that the information from earlier parts of the code is still available (such as the preconditions at the start of the assembly routine and the effects of previous instructions and abstract blocks). If code is divided into multiple program functions this information would likely need to be explicitly restated as preconditions which would result in a larger number of annotations overall.

3.2.2 Ghost code

While dividing the code into blocks helps it is still desirable to keep these blocks as large as possible without having to provide additional annotations such as assertions. Here Why3's concept of *ghost code* [7] is quite useful. Ghost code is code for which Why3 guarantees that it cannot affect the outcome of normal code. This allows us to insert or extend code without the risk of affecting the behavior modeled by the code.

One of the things that ghost code enables is that we can pass values as ghost parameters. An example of a case where this is very helpful is in cases where a consecutive area of main memory is modified. This is often done by storing a pointer to the start of this area in one or multiple registers (depending on whether the pointer fits in a single register), followed by repeated writing to the memory the pointer refers

[6] Here an effect means a modification of the system state. Some examples of effects are: setting the carry flag, loading a constant into a register or storing the contents of a register in main memory (RAM memory).

to and incrementing the pointer. Normally this might result in a list of postconditions like shown in listing 9.

Listing 8: A sequence of 16 `ST Z+, r1` *"Store Indirect From Register to Data Space using Index Z"* instructions modeled in WhyML.

```
label START in
st_inc r30 r1;
label L1 in
st_inc r30 r1;
label L2 in
st_inc r30 r1;
label L3 in
st_inc r30 r1;
      ...
label L15 in
st_inc r30 r1;
label L16
```

Listing 9: Example of what formulas would result from the code from listing 8. The `at` keyword is used here to refer to the state of the main memory and registers at previous points of execution. Why3 itself will not generate formulas containing the `at` keyword. Here it is used to relate the incarnations of `mem` to the original code.

```
(* pointer is stored in registers r30 and r31 *)
(* START and L1 through L16 are code labels (positions in the code) *)
mem at L1 = (mem[(reg[r30] + 256 * reg[r31]) <- reg[r1]]) at START
(reg[r30] + 256 * reg[r31]) at L1 = (1 + reg[r30] + 256 * reg[r31]) at START
mem at L2 = (mem[(reg[r30] + 256 * reg[r31]) <- reg[r1]]) at L1
(reg[r30] + 256 * reg[r31]) at L2 = (1 + reg[r30] + 256 * reg[r31]) at L1
mem at L3 = (mem[(reg[r30] + 256 * reg[r31]) <- reg[r1]]) at L2
(reg[r30] + 256 * reg[r31]) at L3 = (1 + reg[r30] + 256 * reg[r31]) at L2
      ...
mem at L16 = (mem[(reg[r30] + 256 * reg[r31]) <- reg[r1]]) at L15
(reg[r30] + 256 * reg[r31]) at L16 = (1 + reg[r30] + 256 * reg[r31]) at L15
```

If we store the initial pointer value in a ghost variable we can use this variable to define the postconditions in terms of the initial pointer value. The postconditions now could look like what is shown in listing 11.

Listing 10: A sequence of 16 `ST Z+, r1` instructions modeled in WhyML. Here the pointer stored in Z (registers r30 and r31 combined) is passed as a ghost argument to `st_inc'`.

```
label START in
let ghost ptr = uint 2 reg r30 in
st_inc' r30 r1 ptr;
label L1 in
st_inc' r30 r1 (ptr + 1);
label L2 in
st_inc' r30 r1 (ptr + 2);
label L3 in
st_inc' r30 r1 (ptr + 3);
      ...
label L15 in
st_inc' r30 r1 (ptr + 16);
label L16
```

Listing 11: Example of what formulas would result from the code from listing 10. These formulas are easier to work with than the formulas from listing 9 as the pointer values are expressed directly in terms of the original value. Previously we would need to work back through all previous versions of `reg` to see how this value has changed.

```
(* The initial pointer value is stored in ghost variable ptr *)
ptr = reg[r30] + 256 * reg[r31] at START
mem at L1 = (mem[ptr <- reg[r1]]) at START
(reg[r30] + 256 * reg[r31]) at L1 = ptr + 1
mem at L2 = (mem[ptr+1 <- reg[r1]]) at L1
(reg[r30] + 256 * reg[r31]) at L2 = ptr + 2
mem at L3 = (mem[ptr+2 <- reg[r1]]) at L2
(reg[r30] + 256 * reg[r31]) at L3 = ptr + 3
      ...
mem at L16 = (mem[ptr+15 <- reg[r1]]) at L15
(reg[r30] + 256 * reg[r31]) at L16 = ptr + 16
```

Another area where ghost code is quite useful is in using mutable ghost variables to help express what registers were modified. While we can not directly use separate variables to model the registers we can still use them to help express what register got updated in a block of code. By specifying that these ghost variables are synchronized with the modeled registers both before and after execution of a block of code while only updating the ghost variables corresponding to registers that were updated in this block, we can provide the provers with enough information on what registers have changed. An example of this technique in use can be seen in listing 12. While this notation isn't more compact than specifying the same information without ghost registers (see listing 13) this technique seems to generally scale better.

Listing 12: Example of specifying register modifications with shadow registers

```
use avrmodel.Shadow as S

let mul16 (): unit
(* The ghost registers in S.shadow are synchronized with reg.
 * The registers are modeled in reg, which is a mutable map
 * from integers to bytes. *)
  requires { S.synchronized S.shadow reg }

(* uint is a function that returns the value stored in multiple
 * consecutive bytes interpreted as an unsigned integer *)
  ensures { uint 4 reg 12 = old(uint 2 reg 2 * uint 2 reg 7) }

(* After execution the ghost registers are again synchronized with reg *)
  ensures { S.synchronized S.shadow reg }
(* Implicit: writes {reg, S.r0, S.r1, S.r12, etc.} *)
=
  clr r23;
  mul r3 r8;
  movw r14  r0;
  mul r2 r7;
  movw r12 r0;
  mul r2 r8;
  add r13 r0;
  adc r14 r1;
  adc r15 r23;
  mul r3 r7;
  add r13 r0;
  adc r14 r1;
  adc r15 r23;

  (* Update the shadow registers for which the register in reg was updated *)
  S.modify_r0();
  S.modify_r1();
  S.modify_r12();
  S.modify_r13();
  S.modify_r14();
  S.modify_r15();
  S.modify_r23()
```

Listing 13: An alternative way of specifying which registers where modified in mul16.

```
ensures {
reg = (old reg)[r0 <- reg[r0]][r1 <- reg[r1]]
  [r12 <- reg[r12]][r13 <- reg[r13]][r14 <- reg[r14]][r15 <- reg[r15]]
  [r23 <- reg[r23]]
}
```

The assembly code modeled in listing 12 can be seen in listing 14. Observe that the WhyML code looks very much alike.

Listing 14: 16×16 bits multiplication implementation in AVR assembly

```
mul16:  CLR R23
        MUL R3, R8
        MOVW R14, R0
        MUL R2, R7
        MOVW R12, R0
        MUL R2, R8
        ADD R13, R0
        ADC R14, R1
        ADC R15, R23
        MUL R3, R7
        ADD R13, R0
        ADC R14, R1
        ADC R15, R23
```

3.2.3 Underspecification

Another measure that helps to keep the amount of premises in check is to only model those parts of the architecture that are relevant for the modeled code. Parts of the architecture that aren't relevant for the instructions of the modeled code can be safely omitted. For example, when working with AVR code without branching it is safe to omit the zero flag as this part of the state is only used by branch (jump) instructions. Any effects that describe modifications on the modeled state can also be safely omitted, as long as the `writes` specification is complete (for any part X of the architecture model state it is fine to omit information on how an instruction modifies X, but not that it modifies X.). This is equivalent to removing premises in a logic formula: the new formula will only hold if the original formula did.

3.3 Issues with modeling assembly code in Why3

While an implementation of Curve25519 in AVR assembly modeled in WhyML [12, 13] has previously been verified, some difficulties were noticed during this effort.

3.3.1 Type system issues

One of the most noticeable issues is that we previously weren't able to port our work from Why3 0.88 to Why3 1.2 within a reasonable time frame. Why3 1.0 changed how type invariants (constraints placed on the values instances of a type can take) are modeled in its SMT output, which had a dramatic effect on the performance of the provers. Proofs that previously finished within a fraction of a second now no longer succeeded within a minute or ran out of the set 2GB memory limit. While emulating the old behavior by manually modeling type invariants using pre- and postconditions did work, this resulted in at least one additional precondition for each instruction. This greatly increased clutter and would slow (re-)loading files with hundreds of instructions down to such a degree of being unusable. Using range types showed some promise, but it seemed that we still would not be able to match the performance of 0.88 in the number of instructions per annotation. Due to time constraints, we decided to stick with Why3 version 0.88 for [13].

3.3.2 Proof context pollution

The proof context contains all information available to prove the generated proof obligations. The proof context contains all previously defined lemmas and axioms, constants and types as well as the current premises. Examples of the (local) proof

context can be seen in figures 2 through 6: the proof context is the part between `Local Context` and `Goal`. Every annotation or instruction not inside the body of a previous abstract block adds information to the proof context. While this information often is essential, we only need information on the current state and how this relates to the initial state, not all information may be necessary to prove local properties. In section 3.2 some techniques were described to limit the number of premises present in the proof context, but these techniques are not always sufficient. Much of the information is relevant for some of the proof obligations, but not necessarily for the current goal. The proof context may contain so much noise that provers are no longer able to verify the current goal. If the proof context cannot be reduced any further without removing any information necessary to prove later goals it may be useful to model a block of code in a separate program function.

It may be useful to pass some of the registers used as arguments if a step is repeated but some of the registers the step operates on differ. Unfortunately, this comes with disadvantages: where provers generally perform well with registers modeled as static indices, with known values, into a mutable map, they perform far worse if the value of the index isn't known. Additionally, the user needs to manually exclude any register passed as an argument from aliasing with (nearly) any other register in use by the code.

Another disadvantage of dividing code into program functions is that code may be harder to relate to the original assembly code. Why3 doesn't provide a way to recursively replace calls to program functions with the contents of these functions until only calls to the functions modeling instructions are left. This means that checking that a verified program function indeed models a given assembly routine takes an increasing amount of effort the more this routine has been divided into program functions.

3.3.3 Prover dependence

Another issue we hit was that for some goals only a single version of a single prover would be able to prove this goal. Specifically: CVC4 1.4 was often the only prover able to verify that blocks of code performed the specified multiplications. Newer versions of the same prover often wouldn't be able to prove these same goals or would take much longer to do so. While I'm not aware of any soundness issues of CVC4 version 1.4, depending on a single outdated version is something that is best to be avoided.

4 Asm3 framework

To combat the issues listed in section 3.3 the Asm3 framework was created. This framework consists of a plugin for Why3 and it provides a language specializing in modeling assembly code based on Why3's WhyML. The language contains a special feature for modeling statically accessed memory while additionally providing syntax for working with multi-byte integers. The framework provides its own VC generator which features heuristics to automatically filter out most premises that are unlikely to contribute to proving the current proof goal. This filtering step can be configured or disabled per sub-goal.

4.1 The Asm3 language

The Asm3 language is designed to be an annotated variant of traditional assembly source code.The language is similar to WhyML but doesn't have program functions, instead, it has instructions, macros, and entry points.

Instructions are the most basic programming elements in Asm3 and can be used to model assembly instructions. Currently, it is only possible to define instructions axiomatically[7] as it currently is the smallest building block available.

[7]In the future it would be desirable to have the user give a proof of the consistency of the

Macros can be used to model reusable sequences of instructions. The body of a macro consists of a sequence of instructions or invocations of other macros. The arguments for instructions or macro invocations appearing in this body do not have to be constants, but can also be expressed in terms of the arguments of the macro. Like WhyML's program functions macros abstract their implementation details from their users.

Listing 15: Asm3 macro modeling the assembly code from listing 14

```
let macro mul16
ensures { $r12...4 = old ($r2...2 * $r7...2) }
reads { r2, r3, r7, r8 } (* Reads specification is optional here *)
writes { r0, r1, r12, r13, r14, r15, r23 }
writes { cf, hf, sf, vf, nf, zf }
=
  clr r23;
  mul r3 r8;
  movw r14 r0;
  mul r2 r7;
  movw r12 r0;
  mul r2 r8;
  add r13 r0;
  adc r14 r1;
  adc r15 r23;
  mul r3 r7;
  add r13 r0;
  adc r14 r1;
  adc r15 r23
```

Entry points are quite similar to macros in that both model the behavior of sequences of instructions, but unlike macros, entry points cannot be included in another entry point or macro. Where macros model reusable sections of code, entry points are used to model entire assembly programs or routines. Once implemented it would be possible to extract a sequence of instructions and labels by recursively expanding any macros appearing in the body of an entry point. As code extraction currently isn't implemented using entry points does not provide an advantage over using only macros.

4.1.1 Index references

Asm3 contains a concept of *index references* that can be thought of as a hybrid approach between working with Why3's `ref` references and indices into mutable maps. In Asm3 it is possible to use these index references to model memory such as registers using (seemingly) separate references while allowing aliasing and working with offsets. To properly explain what an index reference is and what we can do with it we first need to introduce the related concepts of index types and indexed objects.

To define an index reference we first need to define an *index type*. For example, we can define an index type `i-type` with

```
index i-type = create_index i-obj (a, b)
```

Here `index` and `create_index` are keywords, while `a` and `b` are types. `i-obj` is the name of the memory space indexed by references of index type `i-type`. We will refer to these memory spaces indexed by index references as *indexed objects*.

For simplicity, `i-obj` can be thought of as a mutable variable of type `map a b` while references of index type `i-type` can be thought of as constants of type `a`. `i-obj` never appears in any code or specifications, but may appear in the VCs generated by the Asm3 plugin. It is accessed only through references of index type `i-type`.

specification of an instruction. Users should be able to prove that an implementation of an instruction could exist.

Now we have defined an index type we can define an index reference. We can define such a reference in a similar way to defining a variable, only we don't specify a type, but an index type. For example, we can define an index reference `x` of index type *`i-type`* with `x: i-type`. When `x` is used by itself it is coerced to a value of type `a`. An index reference is generally only useful when used in combination with the `$` operator: the index dereference operator. This operator returns the value stored at the position of an index reference in its corresponding indexed object. For example, we can think of `$x` as returning *`i-obj`*`[x]`.

Listing 16 provides an example in which two index types get defined. The index type `reg` is used to model 32 8-bit wide registers, while the index type `srbit` is used to model 8 bits of the status register SREG. Listing 16 also shows how we can define

Listing 16: A simple example of using index references in the Asm3 language

```
(* Definition of 32 8 bits registers *)
type reg_index : <range 0 31>
(* Definition of index type reg with its indexed object regs *)
index reg = create_index regs (reg_index, byte)
(* variables (x: reg) implicitly index the indexed object regs *)
(* regs is an internal mutable object containing bytes *)

(* Definition of status register SREG, which contains the status flags *)
type rb_index = <range 0 7> (* SREG is 8 bits wide *)
index srbit = create_index sreg (rb_index, bit)

(* Carry flag *)
let constant cf: srbit = 0 (* cf is the value stored in sreg at position 0 *)

(* This instruction models a simplified version of the AVR ADD instruction *)
(* $ dereferences index references *)
val instr add (rd rr: reg)
ensures Post_rd { $rd = mod (old ($rd + $rr)) 256 }
ensures Post_cf { $cf = div (old ($rd + $rr)) 256 }
reads { rd, rr }
writes { rd, cf }
may_alias { rd with rr }

let macro avr_double (dst: reg)
requires { $dst < 128 }
ensures { $dst = 2 * old $dst }
writes {dst, cf}
=
  add dst dst
```

an index reference. The declaration `let constant cf: srbit = 0` defines an index reference `cf`. This reference models the value stored at position 0 in SREG.

Index references are fairly similar in use to Why3's `ref` references. In both cases mutable, memory locations can be modeled with separate references, where modifications are specified in terms of just this reference. (Here memory refers to registers and RAM or any other form of volatile memory.) This allows for a syntax that is very close to that of Why3's `ref` references: if one were to rewrite WhyML code using references to Asm3 code the specifications would need very few changes. The `writes` and `reads` annotations remain unchanged while in the pre- and postconditions the `!ref` dereference operators need to be replaced with `$`. Listings 17, 18 show how the syntax of working with index references compares to working with `ref` in WhyML[8]. Observe that the specifications of simple_add in listings 17 and are nearly identical.

[8]Do note that, although `rd rr : reg` appears in listings 17 and 18 the meaning of `reg` differs.

Listing 17 Asm3: defining a simple addition instruction using index references

```
type reg_index = <range 0 31>
index reg = create_index regs (reg_index, byte)

val instr simple_add (rd rr : reg)
ensures Post_rd { $rd = mod (old ($rd + $rr)) 256 }
reads { rd, rr }
writes { rd }
```

Listing 18 WhyML: defining a simple addition instruction using `ref` references

```
type reg = ref byte

val simple_add (rd rr : reg): unit
ensures Post_rd { !rd = mod (old (!rd + !rr)) 256 }
reads { rd, rr }
writes { rd }
```

Syntax comparison between modeling registers in Asm3 with index references and modeling registers with ref references in WhyML

Unlike WhyML's `ref` references, it is possible to work with offsets. This makes it possible to define instructions that interact with consecutive sections of memory while only passing a single element of this memory. An example of this is given in listing 19, which not only operates on registers `rd` and `rr`, which are given as arguments, but also on the registers next to those registers. Listing 19 also demonstrates why it

Listing 19: Axiomatic defintion of the AVR movw instruction in Asm3

```
val instr movw (rd rr: reg)
writes { rd, rd+1 }
reads { rr, rr+1 }
requires Pre_rd_rr { mod rd 2 = 0 /\ mod rr 2 = 0 }
ensures Post_rd { $rd = old ($rr) }
ensures Post_rdp1 { $(rd+1) = old $(rr+1) }
```

can be useful to be able to interpret index references as a value, rather than just a reference. Here a precondition is given to check that `rd` and `rr` are even as the movw instruction in AVR architecture requires its arguments to be aligned to register pairs.

We have previously demonstrated that it is possible to work with offsets and aliasing in WhyML by modeling memory with a single mutable object. The previous description of index references and indexed objects may lead one to expect that index references just introduce a new syntax for working with a mutable map. That is not the case, however. The Asm3 framework tracks the state of the memory at each position in the indexed object. As long as Asm3 can determine that the memory pointed to by an index reference was last modified from this specific index reference or wasn't modified at all, Asm3 models this memory separately from the rest of the memory modeled by the indexed object. For example, for the following abstract block (abstract blocks are sections of code along with a specification enclosed by the `begin` and `end` keywords):

Listing 20: Abstract block containing two load immidate instructions.

```
begin ensures { $r0 = 37 /\ $r1 = 0 }
```

In listing 18 it denotes a type while in 17 it declares `reg` as a an index into the virtual variable `regs`.

```
  ldi r0 37;
  ldi r1 0
end
```

Asm3 will generate a proof formula that looks similar to:

$$\forall_{\mathrm{r0_{new}},\,\mathrm{r1_{new}}}\ \mathrm{r0_{new}} = 37 \rightarrow \mathrm{r1_{new}} = 0 \rightarrow \mathrm{r0_{new}} = 37 \wedge \mathrm{r1_{new}} = 0$$

Observe that the new values of registers `r0` and `r1` are modeled separately using the variables $\mathrm{r0_{new}}$ and $\mathrm{r1_{new}}$. This is quite different from the proof formula that Why3 would generate if we would model registers with a single mutable map. In that case we would get a formula that would look similar to:

$$\begin{aligned} \forall_{\mathrm{regs_{temp}}} \quad & \mathrm{regs_{temp}} = \mathrm{regs_{old}}[0 \leftarrow 37] \quad \rightarrow \\ \forall_{\mathrm{regs_{new}}} \quad & \mathrm{regs_{new}} = \mathrm{regs_{temp}}[1 \leftarrow 0] \quad \rightarrow \\ & \mathrm{regs_{new}}[0] = 37 \wedge \mathrm{regs_{new}}[1] = 0 \end{aligned}$$

Where $\mathrm{regs_{old}}$ models the initial state of the registers.

The syntax for index references generally leads to more succinct specifications than working with mutable maps. An example of the difference is given in listings 21 and 22. Observe that the postcondition `Post_rd` in the Asm3 version is much shorter than the one in the WhyML version.

Listing 21 Asm3: defining a simple addition instruction using index references

```
type reg_index = <range 0 31>
index reg = create_index regs (reg_index, byte)

val instr simple_add (rd rr : reg)
ensures Post_rd { $rd = mod (old ($rd + $rr)) 256 }
reads { rd, rr }
writes { rd }
may_alias {rd with rr}
```

Listing 22 WhyML: defining a simple addition instruction using mutable maps

```
type reg = <range 0 31>
val regs : ref (map reg byte)

val simple_add (rd rr : reg): unit
ensures Post_rd { !regs = old (!regs[rd <- mod (!regs[rd] + !regs[rr]) 256])  }
reads { regs }
writes { regs }
```

Syntax comparison between modeling registers in Asm3 with index references and modeling registers in WhyML with indices of an integer type into a mutable map

One might wonder why indexed objects are assigned names (such as `regs` and `sreg` in listing 16). The user can only interact with these objects through index references, so why mention these objects at all? The reason is that it isn't always possible to model the values index references point to separately. In cases where index references may be aliased and to one of them an assignment is made these values will be modeled using a single map. To make it clear where this map originates from, this map will appear in VCs as a variable with its name based on the name assigned to the indexed object.

4.1.2 Multi-byte integers

In assembly code it is often necessary to store values with a higher bit size than can be stored at a single memory location, for example in AVR architecture a 32 bits number would need to be stored in 4 separate registers (or other memory locations). In Asm3 it is straightforward to describe values spread over multiple memory locations. It includes an operator for concatenating sized integers: for example, for sized integer expressions[9] `x` and `y`, the expression `x::y` gives a value of `x` and `y` concatenated. This concatenation is defined as $[\![\texttt{x::y}]\!] = [\![\texttt{x}]\!] + \mathit{radix}\,(\texttt{x}) \times [\![\texttt{y}]\!]$, where $[\![e]\!]$ is the semantic value of the Asm3 formula e.

Asm3 also contains syntax for describing values stored in consecutive locations in memory. With the triple dot operator `f(x)...y` one can specify numbers stored over `y` consecutive memory locations. In its current state this is only defined for `$` ($[\![\texttt{\$x...y}]\!] = [\![\texttt{\$x::\$(x+1)::} \cdots \texttt{::\$(x+y-1)}]\!]$) and for the map get operator `[]` ($[\![\texttt{x[z]...y}]\!] = [\![\texttt{x[z]::x[z+1]::} \cdots \texttt{::x[z+y-1]}]\!]$). If modeling a big endian architecture a negative value for `y` can be used to reverse the order of the components of the value. Examples: $[\![\texttt{\$r2...3}]\!] = [\![\texttt{\$r2::\$r3::\$r4}]\!] = [\![\texttt{\$r2}]\!] + 256 \times ([\![\texttt{\$r3}]\!] + 256 \times [\![\texttt{\$r4}]\!])$, $[\![\texttt{\$r4...(-3)}]\!] = [\![\texttt{\$r4::\$r3::\$r2}]\!] = [\![\texttt{\$r4}]\!] + 256 \times ([\![\texttt{\$r3}]\!] + 256 \times [\![\texttt{\$r2}]\!])$, where `r2`, `r3` and `r4` are three consecutive registers.

The `ensures` part of the specification of `movw` given in listing 19 could alternatively be given as `ensures { $rd...2 = old ($rr...2) }`.

4.1.3 Alias annotations

Unlike references in WhyML, whether index references will or will not alias doesn't have to be statically defined. The user only has to specify if indices are allowed to alias. To accomplish this the Asm3 language includes the `may_alias` annotation. With this annotation, the user can specify that in invocations of a macro, given pairs of index object indices may alias. For each invocation of a macro, the framework checks if any aliasing that would occur is allowed. Take the following example:

Listing 23: Example of an procedure in which some arguments may be aliased.

```
let macro add_reg_to_word (dst src tmp: reg)
requires { $dst...2 + $src <= 0xFFFF }
ensures { $dst...2 = old ($dst...2 + $src) }
may_alias { src with tmp }
may_alias { dst, dst+1 with src }
writes {dst, dst+1, tmp}
=
  add dst src;
  clr tmp;
  adc (dst+1) tmp
```

This macro adds `src` to `dst` and adds the carry bit to the register that follows `dst`. `add_reg_to_word r2 r2 r1` would be allowed as while `dst src` both map to `r2` this is allowed by the specification. `add_reg_to_word r2 r2 r2` on the other hand wouldn't be allowed as then `dst tmp` both map to `r2` which isn't allowed according to the specification (`may_alias` is reflexive, but not transitive).

4.1.4 Expressive power of index references

Section 4.1.1 states that index refrences can be thought of as a hybrid approach between WhyML's references and working with mutable maps, but didn't go into detail about what that means.

[9] `x` is positive and has a radix. For for binary values the radix is 2^b where b is the number of bits of the integer size. The radix of a byte is 256, so for any byte i the following holds: $0 \leq i < 256$.

In cases where aliasing cannot occur or cases where no writes occur to indices that can be aliased (with an index passed as argument or a statically defined index), index references can be considered to be a less restrictive version of WhyML's references. Like with WhyML's references, in the generated proof obligations the value stored at the location of each index is modeled with separate variables. Take `add a b (*a := a + b*); assert {$a = 5 /\ $b = 3}` where `a` cannot alias with `b`, the following formula would appear in the obligation generated for this code:

$$a_{\text{new}} = a_{\text{old}} + b_{\text{old}} \rightarrow a_{\text{new}} = 5 \wedge b_{\text{old}} = 3$$

Here a_{old} and b_{old} model the original values of `a` and `b` respectively and variable a_{new} models the new value stored in `a`.

Once a write occurs to an index that may be aliased with another index the modeled memory starts to behave much more similarly to a map. If we again look at `add a b; assert {$a = 5 /\ $b = 3}`, now with the condition that `a` and `b` are allowed to alias.

$$a_{\text{new}} = a_{\text{old}} + b_{\text{old}} \wedge regs_{\text{new}} = regs_{\text{old}}[b \leftarrow b_{\text{old}}][a \leftarrow a_{\text{new}}] \rightarrow a_{\text{new}} = 5 \wedge regs_{\text{new}}[b] = 3$$

Now maps $regs_{\text{new}}$ and $regs_{\text{old}}$ along with locations a and b (for `a` and `b` respectively) appear in the formula to model the new value stored in `b`. The sub-formula $regs_{\text{new}} = regs_{\text{old}}[b \leftarrow b_{\text{old}}][a \leftarrow a_{\text{new}}]$ constructs a new map to model the new value of `b` (Here it is assumed that `b` is not already modeled with $regs_{\text{old}}$). If `a` and `b` are indeed aliases $regs_{\text{new}}[b]$ will be equal to a_{new}, while it will be b_{old} if these indices do not alias. While the new value of `a` is now stored in $regs_{\text{new}}$ we can continue to model a with the new incarnation a_{new} until we would write to `b`.

As soon as a write occurs to an index that could be aliased with other indices these other indices are no longer modeled separately. In these cases Asm3 will construct maps, write the old value stored at the affected indices to it and include the new value of the index that was written to.

Using maps directly is much more powerful, however, as index reference appearing in terms must be statically defined with respect to local (arguments passed to the current macro) or global indices (globally defined indices). This isn't an issue when modeling statically addressed memory such as registers[10], but this makes index references generally unsuitable for modeling main memory. For example, it isn't suitable for modeling a loop that doesn't iterate a predetermined amount of times (the loop can not be unrolled at compile time) that writes to a new location every iteration. An example of this is shown in listing 24 (assume that `n` isn't a constant).

Listing 24: Example of something (written in C) that cannot be modeled with just index references

```
// given arrays x and y, positive integer n
for (int i = 0; i < n; i++) {
    x[i] = y[i];
}
```

The advantages index references have over simple mutable maps in Why3 are that it is easier to specify which specific parts of memory get modified and that values at separate indices can often be modeled in a fully separate way. This last difference can make a large difference in a automatic theorem prover's ability to verify a property in a reasonable amount of time.

4.1.5 Weak types

As previously stated in section 3.3.1 Why3 1.0 changed the way type invariants are modeled in its SMT2 output. This led to an unacceptable loss of prover performance

[10] Architectures exist in which the register being accessed can be determined at run time. Index references aren't a good match for code in which this is prevalent.

when performing a simple syntactic port of our verification work of a curve25519 implementation written in AVR assembly. To be specific: in Why3 0.88 we would model the registers and main memory as an instance of type `address_space`, which was defined as:

```
type address_space = { mutable data: map int int }
  invariant { forall i. 0 <= self.data[i] < 256 }
```

In the SMT2 input it generated for SMT solvers instances of this type were modeled using arrays of integers along with a premise that the invariant held for this instance. In Why3 1.0 and later `address_space` would be modeled as some abstract type with a projection function[11] to an array of integers. Here the type invariant is encoded into the projection function. While this is a perfectly valid way of modeling types with type invariants, the fact that the values are no longer directly modeled but only the projection of these values, had a profound negative effect on the performance of provers used previously.

Asm3 provides an alternative way of modeling types with type invariants. It can model a limited number of type invariants in a similar way as to how it was modeled in earlier versions of Why3. Currently, this is limited to modeling range types, maps containing range types, maps containing maps containing range types, etc.

Asm3 can model the type invariants of these types using only integers and arrays. For example: in both WhyML and Asm3 one could model bytes with `type byte = <range 0 255>`. As previously stated in section 2.2.1 a byte defined this way is not modeled as an integer, but as some abstract type `byte` with an projection function to the integers. In Asm3 bytes can also be defined with `weak_type byte = <range 0 255>`, in which case bytes will be modeled as integers for which the following (automatically generated) predicate applies: `predicate byte'constraint (b: int) = (-1) < b < 256`. We will refer to types modeled this way as *weak types*.

To demonstrate how this difference in modeling bytes translates to different verification conditions it may be useful to have a practical example of the differences. For example, when modeling bytes using a Why3 range type the following VC would get generated for the postcondition of `avr_double` from listing 16:

Listing 25: Proof goal of listing 16 when modeling bytes using Why3s type system

```
constant dst'0 : byte
constant dst'1 : byte

add_Post_dst: byte'int r5'1 = mod (byte'int dst'0 + byte'int dst'0) 256

goal: byte'int dst'1 = 2 * byte'int dst'0
```

If we were to model bytes using a weak type the following VC would get generated:

Listing 26: Proof goal of listing 16 when modeling bytes using weak types

```
constant dst'0 : int
constant dst'1 : int

H1: byte'constraint dst'0
H: byte'constraint dst'1
add_Post_dst: dst'1 = mod (dst'0 + dst'0) 256

goal: dst'1 = 2 * dst'0
```

Note that the new value `dst'1` of register `dst` is described directly while previously only information on `byte'int dst'1` was provided. For some SMT solvers, notably Z3 this can make a significant difference in verification time and/or memory usage. It is worth noting that Why3's IDE offers an option to hide applications of projection function

[11] The projection functions generated by Why3 are not always visible to the user. In some cases the projection function only appears in the input Why3 generates for the provers.

such as `byte'int` that Why3 itself inserted. In this case the formula add_Post_dst as well as the goal formula would *look* identical.

Outside of the generated VCs, weak types have the advantage that less casting functions need to be used, which can make specifications a bit cleaner in cases where Why3 cannot insert the projection function itself.

Weak types come with their own drawbacks: these "types" are not modeled using Why3's type system, meaning that they cannot be used in the definitions of normal types and are not available within logic terms. If bytes are modeled using a weak type, `forall x : byte. f x = y` will be rejected by Why3's type system (it will complain that it doesn't know type byte) however `forall x : int. byte'constraint i -> f (x) = y` will work fine.

4.2 VC generation in the Asm3 framework

Why3 has support for language plugins. These plugins allow users to prove the specifications of code written in languages other than WhyML. Normally this is accomplished by transforming such code into a WhyML AST representation of the code. After parsing the process of generating VCs is identical to that of code written in WhyML. Why3 itself provides plugins to allow working with a small subset of C and Python.

This method wouldn't work well with Asm3, as the Asm3 language doesn't map well to WhyML and Why3's VC generator isn't suitable for applying the optimizations that Asm3 can perform on each sub-goal. Because of this the Asm3 plugin features its own VC generator. A large difference between Why3's VC generator and Asm3's VC generator is that Asm3's VC generator collects information on which premises and variables are likely to be useful for each sub-goal. This information can then be used to generate an optimized VC for each sub-goal. Why3's VC generator, on the other hand, does not keep track of this information and generates a single VC containing the combined proof obligation of all sub-goals.

Unlike Why3, the Asm3 plugin doesn't construct the weakest precondition to generate a VC. Instead, it constructs a set of goal triples $(P,\ I,\ g)$ where g is a sub-goal, P is the set of premises encountered until reaching g and I is the set of index reference incarnations appearing in P and g. This triple is a representation of a formula proving one of the sub-goals appearing in the weakest precondition. For example, $(\{a \vee b, \neg b\},\ \{a, b\},\ a)$ would be a representation of the formula $\forall_{a,b}\,(a \vee b) \wedge \neg b \to a$.

To give a more concrete example: In section 2.2.3 we constructed the weakest precondition for the postcondition of `quadruple_by_ref`:

$$\forall_{b_2}\ \mathit{in\_bounds}(4 \times b_2) \to \mathit{in\_bounds}(b_2 + b_2) \wedge$$
$$(\forall_{b_1}\ b_1 = b_2 + b_2 \to \mathit{in\_bounds}(b_1 + b_1) \wedge (\forall_{b_0}\ b_0 = b_1 + b_1 \to b_0 = 4 \times b_2))$$

Instead of generating a single formula equivalent to this weakest precondition the Asm3 plugin would generate the following set of triples:

$$\left\{ \begin{array}{ccc} \big(\{\mathit{in\_bounds}(4 \times b_2)\}, & \{b_2\}, & \mathit{in\_bounds}(b_2 + b_2)\big), \\ \big(\{\mathit{in\_bounds}(4 \times b_2), b_1 = b_2 + b_2\}, & \{b_1, b_2\}, & \mathit{in\_bounds}(b_1 + b_1)\big), \\ \big(\{\mathit{in\_bounds}(4 \times b_2), b_1 = b_2 + b_2, b_0 = b_1 + b_1\}, & \{b_0, b_1, b_2\}, & b_0 = 4 \times b_2\big) \end{array} \right\}$$

This set represents the following formulas:

$$\forall_{b_2}\ \mathit{in\_bounds}(4 \times b_2) \to \mathit{in\_bounds}(b_2 + b_2)$$

$$\forall_{b_1,b_2}\ \mathit{in\_bounds}(4 \times b_2) \wedge b_1 = b_2 + b_2 \to \mathit{in\_bounds}(b_1 + b_1)$$

$$\forall_{b_0,b_1,b_2}\ \mathit{in\_bounds}(4 \times b_2) \wedge b_1 = b_2 + b_2 \wedge b_0 = b_1 + b_1 \to b_0 = 4 \times b_2$$

Observe that, if these three formulas hold, the weakest precondition must hold and vice versa.

The conjunction of the formulas represented by the sets Asm3's VC generator generates is equivalent to the weakest precondition. The advantage that these goal triples have over the single weakest precondition formula is that it is easier to remove premises and reduce the number of incarnations of index references that need to be modeled. To prove that the formula represented by $(P,\ I,\ g)$ holds it suffices to prove that the one for $(P',\ I',\ g)$ holds, where $P' \subseteq P$ and $I' \subseteq I$ and I' is chosen such that P' and g do not contain free variables. This is useful as P' and I' may be significantly smaller than P and I, which can lead to a significantly faster verification time.

I contains information on which incarnations of other references an index reference incarnation may be aliased with. Using a subset I' instead of I itself may prevent the need to model the effects of aliased writes. For example, take the case where a write occurs to a reference x that may be aliased with a single reference y, but no incarnations of y after this write occur in either P' or g. In this case we do not have to model these incarnations of y and can treat x the same as a reference without possible aliases.

4.2.1 Effect selection

The Asm3 framework can automatically discard premises and variables it deems to be unrelated to the current proof goal. To understand how Asm3 determines which variables and premises are unlikely to be useful we will first look at a naive approach.

Postconditions are formulas that generally relate the new state of the system to a prior state. This means that we will want to pick our premises in such a way that we can see how the *relevant* parts of the state have evolved from the old to the new state. We can expect variables appearing in the current proof goal to be relevant and that for other variables to be relevant it should at least appear in a premise along with a variable determined to be relevant. We can expect relevant premises to contain at least one relevant variable. In other words: we can expect useful premises to be connected to the goal formula via shared variables. For example, if we want to prove $x > 0$ given $x = 5$ and $y = -3$ the first premise is useful as both formulas share variable x while the second premise doesn't provide any information on x.

Lets look at a slightly less trivial example: take the premises for `ensures { !a = 15 }` in listing 27. These premises are listed in table 2 as formulas h1 through h7. Figure 7 shows which premises and variables are connected to the goal term. Using the existence of a path via shared variables as a heuristic yields *some* success: premise h1 and variable c_0 (which models the initial value stored in `c`) aren't connected to the goal formula and can thus be discarded. But clearly we can do better: h4 and h6 also aren't useful in proving $a_2 = 15$. These formulas are connected via b_0, however.

Listing 27: A simple WhyML program function used to demonstrate how the Asm3 plugin filters out irrelevant effects.

```
let silly_program_func (a b c : ref int) : unit
  requires { !a = 3 /\ !b = 6 /\ !c = 0 }
  ensures { !a = 15 }
  ensures { !c = 16 }
=
  c := !b - 2;
  a := 3 * !a;
  c := !c * !c;
  a := !a + !b
```

formula name	formula	variables
h1	$c_0 = 0$	c_0
h2	$b_0 = 6$	b_0
h3	$a_0 = 3$	a_0
h4	$c_1 = b_0 - 2$	b_0, c_1
h5	$a_1 = 3 \times a_0$	a_0, a_1
h6	$c_2 = c_1 \times c_1$	c_1, c_2
h7	$a_2 = a_1 + b_0$	a_1, a_2, b_0
Goal	$a_2 = 15$	a_2

Table 2: Premises and goal formula for `ensures !a = 15` in listing 27

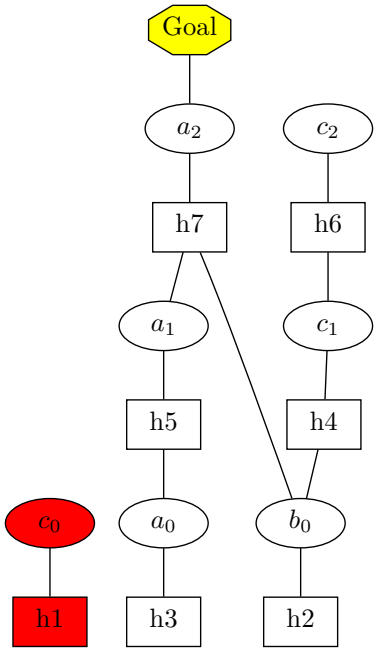


Figure 7: Graph showing which formulas and variables from table 2 are connected to the goal formula. Rectangular nodes represent premises, elliptical nodes represent variables.

We can improve on this naive heuristic by taking into account that not every variable appearing in a formula is alike. Code updates memory so formulas describing the effects of this code likely describe the new values of this updated memory rather than the memory prior to being updated or memory that remained unchanged. Take for example h7 ($a_2 = a_1 + b_0$) which resulted from `a := !a + !b`. We know that of the variables appearing in this statement only `a` is updated, so it is reasonable to assume that this formula provides information on the new value stored in `a`: a_2. The formula is unlikely to provide useful information on the old values stored in `a` and `b`, modeled by a_1 and b_0, as these values were only used and not produced by the statement h7 resulted from.

We should still use the fact that a_1 and b_0 appear in this term, however, as we generally need to use the information from more than a single statement back. We will look at which formulas provide information on these variables and recursively look for formulas providing information on variables appearing in these terms. Using this method we can construct table 3 and its corresponding directed graph in figure 8. Observe that premises h4 and h6 cannot be reached from the goal and should be discarded according to this method.

formula name	formula	Include information on	Provides information on
h1	$c_0 = 0$		c_0
h2	$b_0 = 6$		b_0
h3	$a_0 = 3$		a_0
h4	$c_1 = b_0 - 2$	b_0	c_1
h5	$a_1 = 3 \times a_0$	a_0	a_1
h6	$c_2 = c_1 \times c_1$	c_1	c_2
h7	$a_2 = a_1 + b_0$	a_1, b_0	a_2
Goal	$a_2 = 15$	a_2	

Table 3: Table 7 extended with information on direction of information.

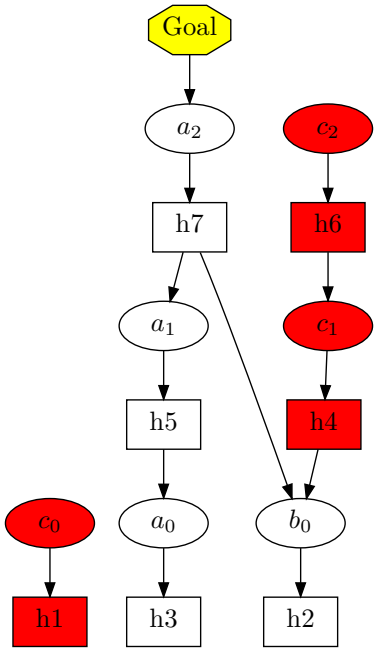


Figure 8: Graph showing which formulas and variables from table 3 can be reached from the goal term

While this heuristic can be a bit overaggressive with removing premises[12], it generally performs very well on filtering out irrelevant effects of code. The Asm3 framework uses this heuristic to filter premises for each proof goal. For each statement or block of code, it combines the `writes` specification of this code with the variables appearing in the postcondition formulas to determine on which variables these formulas provide information. Asm3 then constructs a dependency graph using this information and uses this graph to determine which premises and variables are relevant. Premises can also be explicitly included by annotating the assertion or postcondition with `using` *premise-name*.

The user can disable Asm3's automatic premise selection for specific goals by annotating these assertions with the attribute `asm3vc:filter_premises:off`.

4.2.2 Predictable premise names

To improve readability as well and make it easier for users to manually include or remove premises the Asm3 plugin assigns names to premises resulting from postcon-

[12]It assumes that postconditions do not provide useful information on the old state of the system and will fail to include formulas containing only variables bound by quantifiers.

ditions. These names are prefixed with the name of the instruction used or macro invoked, followed by the name of the postcondition and the position of this invocation within the macro or entry point. For example, if a macro contains four instruction/-macro invocations and the second invocation is a invocation of `add` this will result in a premise named `add_Post_rd_at_2`, if add has a postcondition named `Post_rd`.

As long as the macro contains the same invocations in the same order, the premises generated by these calls will always have the same name. This means that the user can add assertions or divide the code with abstract blocks without fear of proofs breaking because the Why3's `remove` transformation suddenly removed the wrong premise.

5 Comparison with WhyML: 16×16 multiplication

We will take a look at how modeling assembly code in the Asm3 language compares to modeling assembly code in WhyML. Here we will take a look at the difference between the Asm3 code and the WhyML code as well as the difference in the VCs generated for this code. We will use the the 16×16 bits multiplication example code from listing 14 as an example to illustrate this difference.

In listing 28 the 16×16 bits multiplication routine is modeled in the Asm3 language. The code is broken up into two macros: the macro `mul16` models the entire multiplication routine while `mul_add_to_3_regs` models a step that is performed twice in this routine. `mul16` also contains an abstract block (the code enclosed by the `begin` and `end` keywords) to abstract away from the implementation details of a section of the code. While it isn't necessary to break this code up into small sections in order to prove the specification of this code, it helps to illustrate that Asm3 specifications are generally less verbose and results in small VCs.

Listing 28: 16×16 multiplication modeled using two macros

```
let macro mul_add_to_3_regs (dst a b zr: reg)
  requires zr_is_0 { $zr = 0 }
  (* Only define for cases where an overflow can't occur *)
  requires result_fits { $dst...3 + $a * $b <= 0xffffff }
  ensures { $dst...3 = old (($dst...3) + $a * $b) }
  writes { r0, r1, dst, dst+1, dst+2 }
  writes { cf, hf, sf, vf, nf, zf }
  reads { a, b, zr, dst, dst+1, dst+2 }
  may_alias { a, b with b, dst, dst+1, dst+2 }
=
  mul a b;
  add dst r0;
  adc (dst+1) r1;
  adc (dst+2) zr

let macro mul16
ensures { $r12...4 = old ($r2...2 * $r7...2) }
writes { r0, r1, r12, r13, r14, r15, r23 }
writes { cf, hf, sf, vf, nf, zf }
reads { r2, r3, r7, r8 }
=
  clr r23;
begin ensures { $r12...4 = $r2 * $r7 + 0x010000 * ($r3 * $r8) }
  mul r3 r8;
  movw r14 r0;
  mul r2 r7;
  movw r12 r0;
end;
  mul_add_to_3_regs r13 r2 r8 r23;
  mul_add_to_3_regs r13 r3 r7 r23
```

5.1 Comparing specification annotations

Listing 29 shows a WhyML definition equivalent to the formalization of the 16×16 multiplication given in listing 28. Observe that the WhyML version requires an additional `ensures` annotation in the abstract block. The `ensures` annotations specifying the writes in the WhyML version are also harder to read than the writes annotations in the Asm3 version. The `ensures` annotations limiting aliasing are harder to get right than the `may_alias` annotations in WhyML. If the user forgets[13] that some arguments might alias the Asm3 specification is still correct (albeit a bit weaker). While if the user forgets to specify that arguments are not allowed to alias in the WhyML version, the specification is incorrect. For example, if the user forgets to specify that `dst` must not alias with `r0` or `r1` in `mul_add_to_3_regs` (WhyML), then the current postcondition cannot be proven.

Listing 29: WhyML version of the code from listing 28

```
let mul_add_to_3_regs (dst a b zr: reg): unit
  requires zr_is_0 { !regs[zr] = 0 }
  (* Only define for cases where an overflow can't occur *)
  requires result_fits { uint3b !regs dst + !regs[a] * !regs[b] <= 0xffffff }
  requires { 30 > dst > 1 }
  requires { dst <> zr /\ dst+1 <> zr /\ zr > 1 }
  ensures { uint3b !regs dst = old (uint3b !regs dst + !regs[a] * !regs[b]) }
}
ensures {
    let nregs = !regs in
    let oregs = !(old regs) in
    let dst_p1 = reg_add_int dst 1 in
    let dst_p2 = reg_add_int dst 2 in
    nregs = oregs[r0 <- nregs[r0]]
      [r1 <- nregs[r1]]
      [dst <- nregs[dst]]
      [dst_p1 <- nregs[dst_p1]]
      [dst_p2 <- nregs[dst_p2]]
}
  writes { regs }
  writes { cf, hf, sf, vf, nf, zf }
  reads { regs }
=
  mul a b;
  add dst r0;
  adc (dst+(1:reg)) r1;
  adc (dst+(2:reg)) zr

let mul16 (): unit
ensures {
  uint4b !regs r12 =
  old ((!regs[r2] + 256 * !regs[r3]) * (!regs[r7] + 256 * !regs[r8]))
}
ensures Reg_writes {
    let nregs = !regs in
    let oregs = !(old regs) in
    nregs = oregs
      [r0 <- nregs[r0]]
      [r1 <- nregs[r1]]
      [r12 <- nregs[r12]]
      [r13 <- nregs[r13]]
      [r14 <- nregs[r14]]
      [r15 <- nregs[r15]]
```

[13] In fact, the specification given in 28 is weaker than it could have been as `a` or `b` aliasing with `r0` or `r1` would still be safe, but isn't included in the specification.

```
      [r23 <- nregs[r23]]
}
writes { regs }
writes { cf, hf, sf, vf, nf, zf }
reads { regs }
=
  clr r23;
begin ensures {
  uint4b !regs r12 =
  old (!regs[r2] * !regs[r7] + 0x010000 * (!regs[r3] * !regs[r8]))
}
ensures {
    let nregs = !regs in
    let oregs = !(old regs) in
    nregs = oregs
      [r0 <- nregs[r0]]
      [r1 <- nregs[r1]]
      [r12 <- nregs[r12]]
      [r13 <- nregs[r13]]
      [r14 <- nregs[r14]]
      [r15 <- nregs[r15]]
}
  mul r3 r8;
  movw r14 r0;
  mul r2 r7;
  movw r12 r0;
end;
  mul_add_to_3_regs r13 r2 r8 r23;
  mul_add_to_3_regs r13 r3 r7 r23
```

Listing 30: Definitions for type reg and several functions used in listing 29

```
type reg = uint5 (* where uint5 is defined as a 5 bit unsigned integer *)

(* Addition is only defined for int *)
val function reg_add_int (x : reg) (y : int): reg
requires { -1 < uint5'int x + y < 256 }
ensures { uint5'int result = uint5'int x + y }

function uint3b (regs: map reg byte) (start : reg): int =
  regs[start] + 256 *
    (regs[reg_add_int start 1] + 256 * regs[reg_add_int start 2])
function uint4b (regs: map reg byte) (start : reg): int =
  regs[start] + 256 *
    (regs[reg_add_int start 1] + 256 * (
      regs[reg_add_int start 2] + 256 * regs[reg_add_int start 3]
    ))
```

5.2 Comparing VCs

To illustrate the difference in size between the VCs generated for the Asm3 code and the WhyML code we will look at the VCs generated for `mul_add_to_3_regs`. The behavior of the instructions used in the multiplication routine have been fully modeled[14]. While it isn't necessary to model the full behavior of these instructions in this case, it should provide a good approximation of real-world usage of using WhyML or Asm3 to prove the specification of existing assembly code. It allows for an automatic translation of straight-line assembly code to Asm3 or WhyML and doesn't require the

[14] With the exception of the effects on the the program counter.

user to know up front what behavior is relevant of the modeled code. The axiomatic definitions of the instructions used by the Asm3 code are listed in appendix 5.

During the generation of the VC for the postcondition `ensures { $dst...3 = old (($dst...3) + $a * $b) }` of `mul_add_to_3_regs` the Asm3 plugin can discard a number of effects and variable incarnations. Figure 9 shows how the effects and variable incarnations are connected in the dependency graph the Asm3 plugin generates for this VC. The effects and variable incarnations that aren't considered to be useful by Asm3's effect selection heuristic are marked red. Observe that the majority of the effects is not considered to be useful.

Figure 10 shows the VCs (after introducing the premises and variables) generated for both the WhyML and Asm3 version side by side. Here we can see that the VC generated for the WhyML version we can see that the Asm3 version has a significantly smaller proof context as a result of the effect selection optimization.

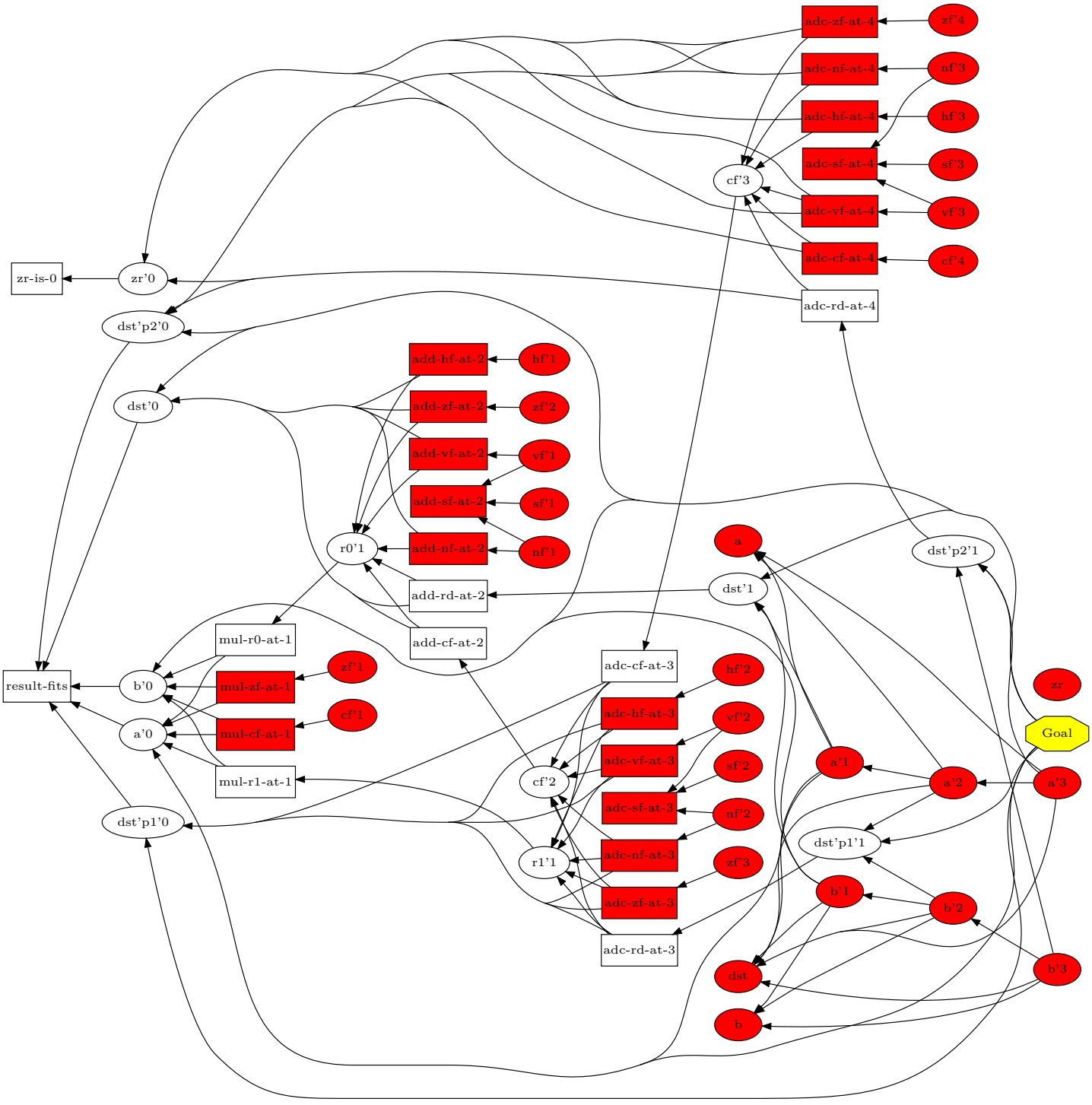


Figure 9: Dependency graph generated by Asm3 for the postcondition of `mul_add_to_3_regs` (node labels shortened for readability). This graph has been generated by the Asm3 plugin.

```
constant regs4 : uint5 -> uint8

constant dst : uint5

constant a : uint5

constant b : uint5

constant zr : uint5

zr_is_0 : regs4[zr] = 0

H4 : 30 > dst

H3 : dst > 2

H2 : not dst = zr

H1 : not (dst + 1) = zr

H : zr > 1

result_fits : (uint3b regs4 dst + (regs4[a] * regs4[b])) <= 0xFFFFFF

constant zf3 : bit

constant cf3 : bit

constant regs3 : uint5 -> uint8

Ensures9 : regs3 = regs4[r0 <- regs3[r0]][r1 <- regs3[r1]]

Ensures8 :
  let result = regs4[a] * regs4[b] in
  regs3[r0] = mod1 result 256 /\
  regs3[r1] = div1 result 256 /\
  cf3 = div1 result 0x8000 /\ zf3 = (if result = 0 then 1 else 0)

constant hf2 : bit

constant sf2 : bit

constant vf2 : bit

constant nf2 : bit

constant zf2 : bit

constant cf2 : bit

constant regs2 : uint5 -> uint8

Ensures7 : regs2 = regs3[dst <- regs2[dst]]

Ensures6 :
  let result = regs3[dst] + regs3[r0] in
  regs2[dst] = mod1 result 256 /\
  regs2 = regs3[dst <- regs2[dst]] /\
  cf2 = div1 result 256 /\
  hf2 = div1 (mod1 regs3[dst] 16 + mod1 regs3[r0] 16) 16 /\
  sf2 = mod1 (vf2 + nf2) 2 /\
  nf2 = div1 (mod1 result 256) 0x80 /\
  zf2 = (if mod1 result 256 = 0 then 1 else 0) /\
  vf2
  = (if regs3[dst] >= 128 /\ regs3[r0] >= 128 /\ not mod1 result 256 >= 128 \/
        not regs3[dst] >= 128 /\
        not regs3[r0] >= 128 /\ mod1 result 256 >= 128
     then 1 else 0)

constant o1 : uint5

Ensures5 : o1 = (dst + 1)

constant hf1 : bit

constant sf1 : bit

constant vf1 : bit

constant nf1 : bit

constant zf1 : bit

constant cf1 : bit

constant regs1 : uint5 -> uint8

Ensures4 : regs1 = regs2[o1 <- regs1[o1]]

Ensures3 :
  let result = (regs2[o1] + regs2[r1]) + cf2 in
  regs1[o1] = mod1 result 256 /\
  cf1 = div1 result 256 /\
  hf1 = div1 (mod1 regs2[o1] 16 + mod1 regs2[r1] 16) 16 /\
  sf1 = mod1 (vf1 + nf1) 2 /\
  nf1 = div1 (mod1 result 256) 0x80 /\
  zf1 = (if mod1 result 256 = 0 then 1 else 0) /\
  vf1
  = (if regs2[o1] >= 128 /\ regs2[r1] >= 128 /\ not mod1 result 256 >= 128 \/
        not regs2[o1] >= 128 /\
        not regs2[r1] >= 128 /\ mod1 result 256 >= 128
     then 1 else 0)

constant o : uint5

Ensures2 : o = (dst + 2)

constant hf : bit

constant sf : bit

constant vf : bit

constant nf : bit

constant zf : bit

constant cf : bit

constant regs : uint5 -> uint8

Ensures1 : regs = regs1[o <- regs[o]]

Ensures :
  let result = (regs1[o] + regs1[zr]) + cf1 in
  regs[o] = mod1 result 256 /\
  cf = div1 result 256 /\
  hf = div1 (mod1 regs1[o] 16 + mod1 regs1[zr] 16) 16 /\
  sf = mod1 (vf + nf) 2 /\
  nf = div1 (mod1 result 256) 0x80 /\
  zf = (if mod1 result 256 = 0 then 1 else 0) /\
  vf
  = (if regs1[o] >= 128 /\ regs1[zr] >= 128 /\ not mod1 result 256 >= 128 \/
        not regs1[o] >= 128 /\ not regs1[zr] >= 128 /\ mod1 result 256 >= 128
     then 1 else 0)

------------------------------ Goal ------------------------------

goal mul_add_to_3_regs'vc :
  uint3b regs dst = (uint3b regs4 dst + (regs4[a] * regs4[b]))
```

VC generated for WhyML version

```
constant zr'0 : byte

constant cf'2 : bit

constant cf'3 : bit

constant r1'1 : byte

constant r0'1 : byte

constant dst'p2'0 : byte

constant dst'p2'1 : byte

constant dst'p1'0 : byte

constant dst'p1'1 : byte

constant dst'0 : byte

constant dst'1 : byte

constant b'0 : byte

constant a'0 : byte

zr_is_0 : zr'0 = 0

result_fits :
  ((dst'0 + (256 * (dst'p1'0 + (256 * dst'p2'0)))) + (a'0 * b'0)) <= 0xFFFFFF

mul_Post_r0_at_1 : r0'1 = mod (a'0 * b'0) 256

mul_Post_r1_at_1 : r1'1 = div (a'0 * b'0) 256

add_Post_rd_at_2 : dst'1 = mod (dst'0 + r0'1) 256

add_Post_cf_at_2 : cf'2 = div (dst'0 + r0'1) 256

adc_Post_rd_at_3 : dst'p1'1 = mod ((dst'p1'0 + r1'1) + cf'2) 256

adc_Post_cf_at_3 : cf'3 = div ((dst'p1'0 + r1'1) + cf'2) 256

adc_Post_rd_at_4 : dst'p2'1 = mod ((dst'p2'0 + zr'0) + cf'3) 256

------------------------------ Goal ------------------------------

goal mul_add_to_3_reg_VC_1'vc :
  (dst'1 + (256 * (dst'p1'1 + (256 * dst'p2'1))))
  = ((dst'0 + (256 * (dst'p1'0 + (256 * dst'p2'0)))) + (a'0 * b'0))
```

VC generated for Asm3 version

Figure 10: Proof context comparison for mul_add_to_3_regs

6 Real world example: 64×64 Karatsuba multiplication

To demonstrate the effects of the optimizations the Asm3 plugin can apply during VC generation. I have ported the formalization of the AVR implementation of 64×64 Karatsuba multiplication from [12, 13] to Asm3[15]. Like the WhyML formalization this is based on, only a part of the AVR architecture is modeled. Specifically, six of the special register (SREG) flags as well as the stack have been omitted and only the instructions used in this code have been modeled. The resulting Asm3 code is included in appendix C. This assembly routine was chosen as a benchmark to demonstrate that the Asm3 language and VC generator combination can be applied to model assembly routines of at least a few hundred instructions in length. This assembly routine consists of 286 instructions and is modeled as one continuous section of code.

First the formalization of the 64×64 Karatsuba multiplication routine was ported from WhyML to Asm3. Next assertions were added or modified and the abstract block specifications were adjusted until all resulting VCs could be verified when all of Asm3's VC optimizations are enabled. Next the verification time for each proof goal of a selection of provers were recorded[16]. This step was repeated for any combination of tested optimizations.

The verification times were recorded for the following optimization combinations:

1. Effect selection enabled and independent (no aliasing can occur or neither has been written to this point) index references modeled with separate variables.

2. Effect selection disabled and independent index references modeled separately.

3. Effect selection disabled and index references modeled as indices into shared maps. In this case index references would provide little more than syntactic sugar for working with maps. The VCs generated by the Asm3 plugin are similar to those generated by Why3.

Additionally the effect of modeling bytes with a weak type as opposed to modeling bytes with a Why3 range type was tested for each combination, giving us six total combinations. Applying the effect selection optimization without modeling independent index references separately isn't implemented currently, so unfortunately this couldn't be tested separately.

The following provers were used to measure the effects of the optimizations on verification performance: versions 4.8.9 and 4.4.1 of Z3 [5], versions 1.7 and 1.4 of CVC4 [1], versions 2.0.0 and 2.3.3 of Alt-Ergo [4], CVC3 version 2.3 and version 2.3 of the E prover [15]. Multiple versions of the same prover were used as some previous proofs in [13] depended on a specific version of a prover. Because of this I thought it relevant to check whether multiple versions of the same prover respond similarly to the same optimizations.

We will consider proof goals generated for assertions and instruction preconditions separately. The 32 preconditions of the load and store instructions used are fairly similar. As these instructions are used in three separate sections of the code, we can use the verification time of these preconditions to get an idea of how the optimizations affect how well the method scales to increasingly large code sizes. The 39 goals resulting from the `assert` and `ensures` annotations give an indication of how the optimizations affect more challenging verification goals.

[15] The resulting code isn't a 1:1 port, most notably some postconditions of the abstract blocks have been changed.

[16] Excluding goals that could be verified using the compute transformation.

6.1 Effects of each configuration on verification of assertions and postconditions

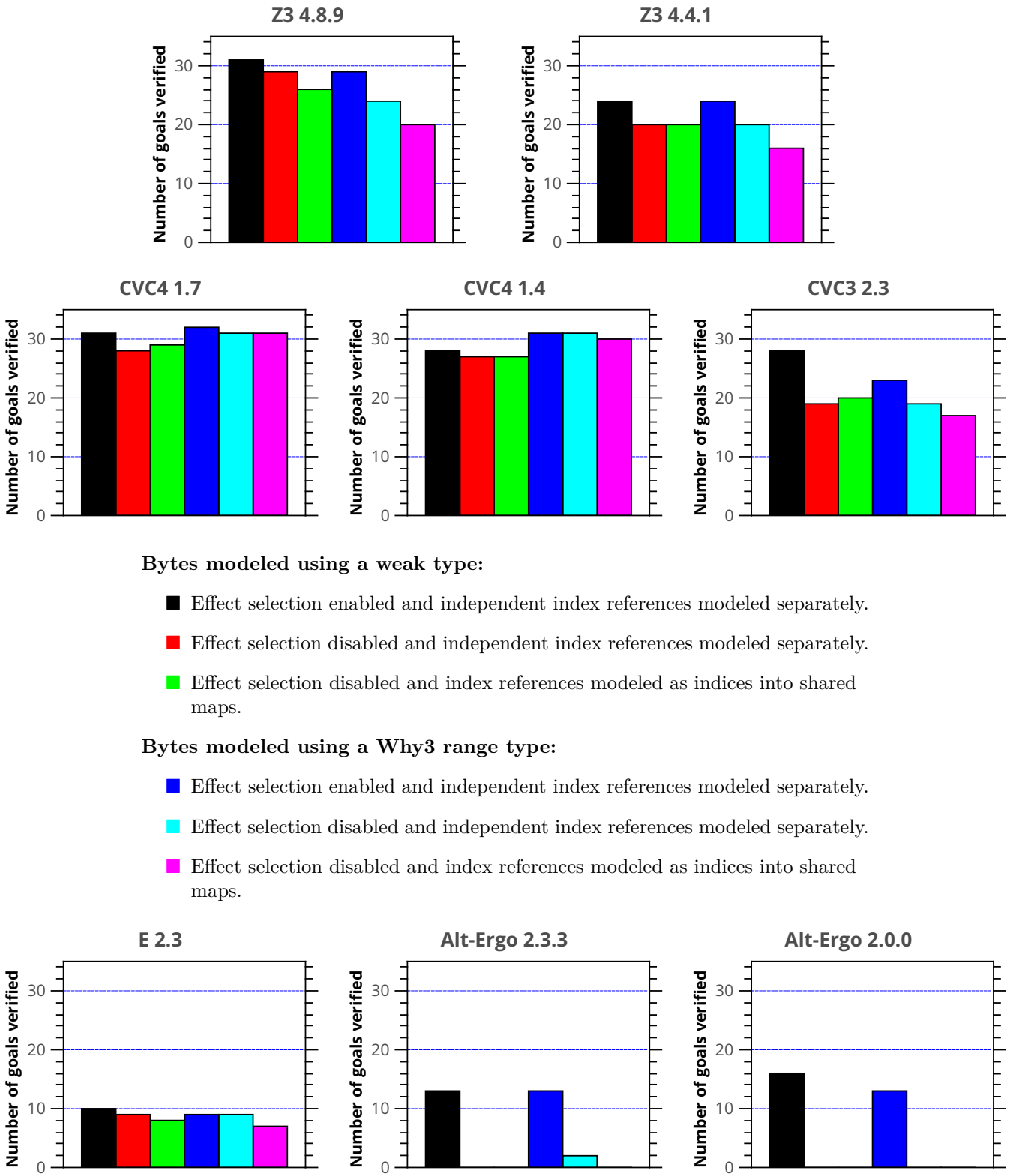
We will first look at the effects of each optimization level on the number of proof goals that each prover was able to verify. Figure 11 shows this number for each of the provers. Having both the effect selection optimization enabled and modeling index references separately always led to the highest number of verified goals. Whether modeling bytes using a weak type or using a why3 range type led to better performance depends on the prover. Z3, CVC3, alt-ergo 2.0.0 and the E prover performed better when bytes were modeled using a weak type, while the reverse is true for CVC4. Modeling index references with separate variables instead of with shared maps usually led to better results. However, when looking at CVC4 1.7 and CVC3 and modeling bytes using weak types this led to slightly worse results if no premises get discarded.

Of course the total number of verified goals, doesn't paint a complete picture. The provers were able to verify different goals depending on which optimizations were used. If bytes are modeled using a weak type both versions of CVC4 can prove two goals that could not be proven if bytes are modeled using a Why3 range type. However, in that case CVC4 1.7 can prove three additional goals and CVC4 1.4 five additional goals. Z3 4.4.1 could prove 2 goals only if index references were modeled as indices into maps. No other prover could verify goals that couldn't be verified when the effect selection was optimization enabled and index references modeled using separate variables.

6.1.1 Verification time

Figure 12 shows verification speed compared to median verification speed for all six configurations. The results for Alt-Ergo aren't shown as both versions were only usable when both the effect selection and aliasing optimizations were enabled.

On average, modeling bytes using a weak type instead of using Why3's own range type led to shorter verification times for both versions of Z3 and version 1.4 of CVC4 while other provers showed no significant difference. Disabling the optimizations Asm3 offers nearly universally led to longer verification times. The way in which bytes were modeled made a significant difference here. For the two Z3 versions as well as CVC4 version 1.4, disabling optimizations had a much more profound effect on verification times if bytes were modeled using Why3's own range type. For CVC3 and CVC4 1.7 the verification times increased much more if bytes were modeled using a weak type. A number of goals was impacted especially severely with, for example, the verification the postcondition `Compute_L` by Z3 4.8.9 jumping from 11.68 to 75.68 seconds when disabling the effect selection optimization. Another striking example is that the verification by CVC4 1.7 of the two goals generated for the assertion `tf_abs_property` jumps from 0.12 seconds to 28.87 seconds total when disabling this optimization (bytes modeled using Why3 range type).



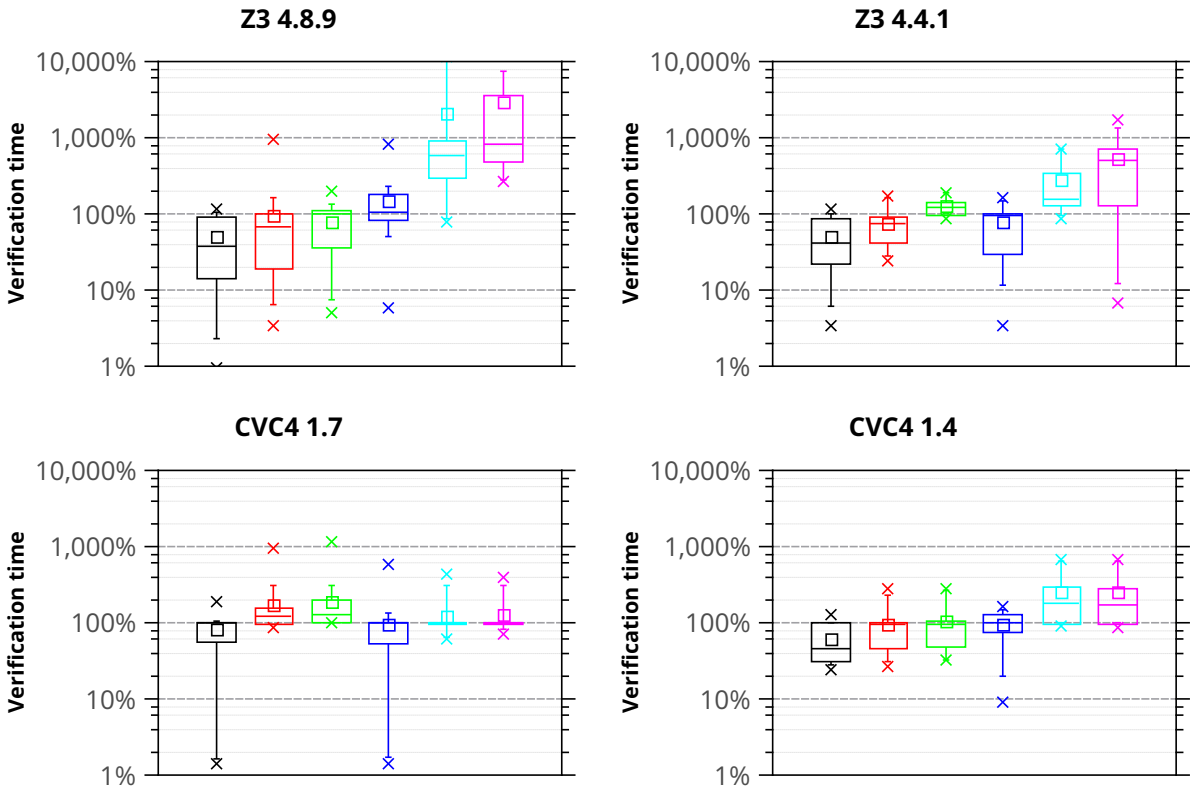
Bytes modeled using a weak type:

- Effect selection enabled and independent index references modeled separately.
- Effect selection disabled and independent index references modeled separately.
- Effect selection disabled and index references modeled as indices into shared maps.

Bytes modeled using a Why3 range type:

- Effect selection enabled and independent index references modeled separately.
- Effect selection disabled and independent index references modeled separately.
- Effect selection disabled and index references modeled as indices into shared maps.

Figure 11: Effect of Asm3's VC optimizations on the number of goals proven. These goals are generated for assertions and postconditions.



Bytes modeled using a weak type:

- Effect selection enabled and independent index references modeled separately.
- Effect selection disabled and independent index references modeled separately.
- Effect selection disabled and index references modeled as indices into shared maps.

Bytes modeled using a Why3 range type:

- Effect selection enabled and independent index references modeled separately.
- Effect selection disabled and independent index references modeled separately.
- Effect selection disabled and index references modeled as indices into shared maps.

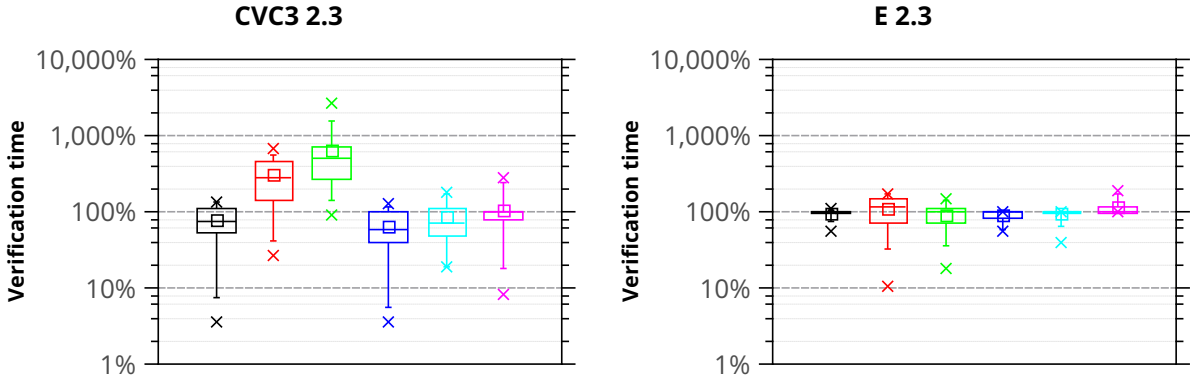


Figure 12: Box plots showing the effect of Asm3's VC optimizations on the verification time relative to the median verification time of that goal.

Prover	# goals	All optimizations enabled	Effect selection disabled	All optimizations disabled
Z3 4.8.9	26	2.49s	9.75s	10.99s
Z3 4.4.1	17	9.63s	22.11s	26.09s
CVC4 1.7	28	4.88s	19.48s	21.52s
CVC4 1.4	27	10.31s	9.83s	9.38s
CVC3 2.3	19	11.99s	44.3s	69.01s
E 2.3	8	10.72s	16.02s	10.39s

Table 4: Comparison of cumulative verification time of goals that could be verified with all optimization levels (bytes modeled with a weak type)

Prover	# goals	All optimizations enabled	Effect selection disabled	All optimizations disabled
Z3 4.8.9	20	11.55s	34.75s	142.45s
Z3 4.4.1	14	8.41s	22.81s	104.17s
CVC4 1.7	31	10.65s	55.99s	53.35s
CVC4 1.4	30	29.34s	48.24s	45.1s
CVC3 2.3	13	5.97s	13.04s	12.19s
E 2.3	6	7.82s	7.29s	18.92s

Table 5: Comparison of cumulative verification time of goals that could be verified with all optimization levels (bytes modeled with Why3 range type)

6.2 Effects of each configuration on verification of preconditions

Figure 13 shows the change in verification time of the preconditions of the three sections of load and store instructions when changing which optimizations are used (See lines 231-238, 306-320 and 659-670 of appendix C for these sections of load and store sections). For these proof goals having the effect selection optimization enabled results in little change in verification speed over the span of the code. However, without this optimization the verification time increases greatly as the proof context grows. Modeling index reference incarnations using shared maps also resulted in an increase in verification time, albeit less severe. Both versions of Alt-Ergo as well as CVC3 were no longer able to verify the VCs generated for the load and store instructions past instruction 274 when the effect selection optimization wasn't enabled. Z3 didn't suffer as much without this optimization, but wouldn't verify any of the VCs of this category if index reference incarnations were modeled using shared maps. Alt-Ergo also did not perform well in that case but could still verify the preconditions of instructions 4 up to and including 11. The E prover wasn't able to verify any of the VCs generated for instruction preconditions.

6.3 Summary

Asm3's optimizations in general have a large positive influence on the number of goals each prover can verify as well as the time that is needed to verify these goals. Z3, CVC3, and Alt-Ergo especially benefit from the optimizations. When modeling bytes using a weak type instead of a why3 range type and applying Asm3's VC generation optimizations, the total number of proof goals Alt-Ergo 2.3.3 was able to verify increased from 8 to 45 goals, for Z3 4.8.9 this increased from 20 to 63 and for CVC3 this increased from 37 to 60. For CVC4 the difference was mainly expressed in a significantly shorter verification time.

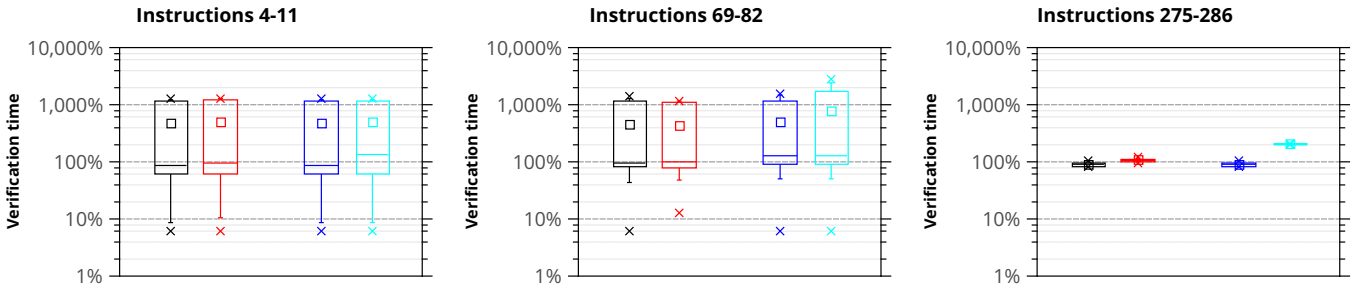
When using the optimizations every goal was provable by at least two provers, counting different prover versions as separate provers and excluding Alt-Ergo 2.0.0 due to this version containing a soundness error. Every goal was provable without using the relatively old CVC3 2.3, CVC4 1.4, and Z3 4.4.1 provers and when excluding these older provers 74% of the goals generated for the assertions and postconditions could be proven by at least two provers. Without Asm3's optimizations this number drops to just 54%, even worse only 90% of the goals were verified at all. From this we can conclude that the effect selection optimization combined with the use of index references generally results in more efficient VCs.

Note that the performance numbers of the provers given in this thesis shouldn't be used to gauge the quality of each prover. For example, with some modifications such as adding additional assertions it should be possible to increase the percentage of goals that most provers can verify. Furthermore different provers tend to have different areas in which they perform well. Being able to use a wide selection of provers is one of Why3's main advantages over similar platforms.

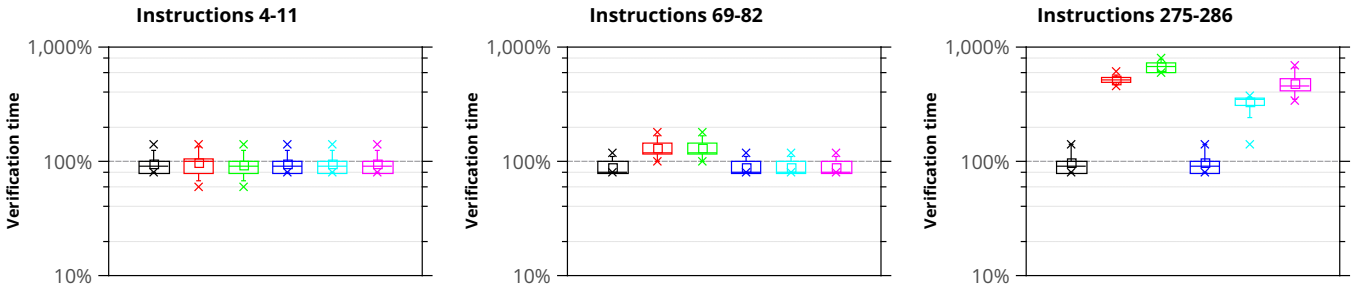
6.4 Comparison with prior verification work on this assembly routine

When compared to the verification from [12] of the 64×64 Karatsuba multiplication routine the total number of used annotations has remained similar. In this prior article 21 annotations were used to prove the specification of this routine, while in the Asm3 version 19 annotations are used[17] (22 if counting the `eliminate_if` transformations

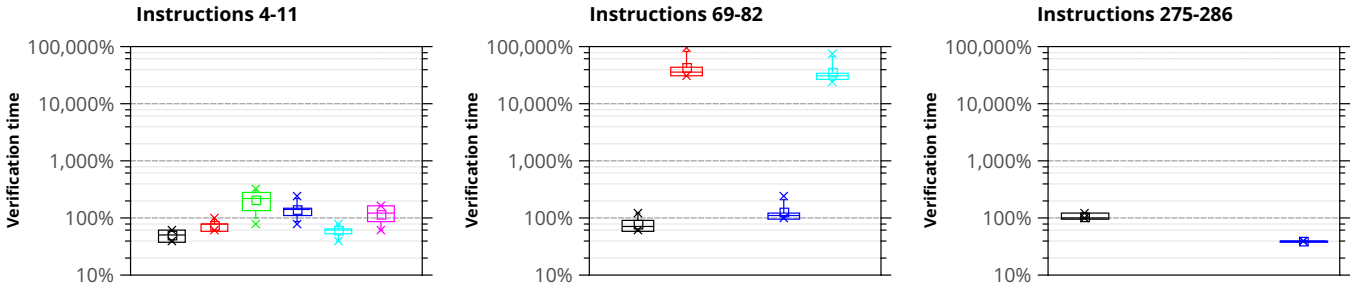
[17] This number could be reduced, but this results in longer verification times and a lower average number of provers able to prove a goal. Additionally, I do not expect real-world usage to include a process of reducing the number of annotations after having succeeded in verifying that the specification is correct.



(a) Z3 4.8.9



(b) CVC4 1.7



(c) Alt-Ergo 2.3.3

Bytes modeled using a weak type:

- Effect selection enabled and independent index references modeled separately.
- Effect selection disabled and independent index references modeled separately.
- Effect selection disabled and index references modeled as indices into shared maps.

Bytes modeled using a Why3 range type:

- Effect selection enabled and independent index references modeled separately.
- Effect selection disabled and independent index references modeled separately.
- Effect selection disabled and index references modeled as indices into shared maps.

Figure 13: Effect of Asm3's VC optimizations on the verification time of instruction preconditions relative to the median verification time of all instruction preconditions.

used among the annotations). The minimum verification time when using the provers used in [12] (CVC4[18], CVC3, E) was lower: 19 seconds instead of 96 seconds. This time was achieved on slightly faster hardware, however. When also using both version of Z3 as well as Alt-Ergo 2.3.3 this time could be further reduced to 16 seconds (12 seconds if cherry picking the fasted time for each goal across all tested optimization combinations).

My prior port of this formalization to Why3 1.2 resulted in similar results as when Asm3's optimizations are not enabled. The Asm3 version features a much faster (re-)load time however as the number of (sub-)goals is significantly lower.

7 Future work

While the current Asm3 framework is functional, it has some significant limitations.

7.1 Integration into Why3

Currently, the features the Asm3 plugin provides are built on top of Why3 rather than integrated into it. This wouldn't be a problem if this plugin simply performed a syntax level translation from the Asm3 language to WhyML, but instead it generates its own VCs. This is undesirable because of several reasons. Most importantly this leads to a larger than necessary trusted code base size as both Asm3's VC generator and Why3's own VC generator need to be trusted. Aside from an increased likelihood of bugs this also leads to an increased maintenance burden.

Another issue is that Asm3's VC optimizations cannot be performed as normal goal transformations. In Why3 it is possible to modify a verification condition using transformations, for example a transformations exists to split the current goal at conjunctions, perform a case distinction or apply the default rewrite rules. Whether or not Asm3 filters out premises can only be controlled via annotations in the Asm3 code. Another thing that can be considered to be an optimization, using weak types instead of Why3's own way of modeling type invariants can only be controlled by defining something as either a weak type or a type. Aside from being undesirable from a consistency standpoint, this means that it currently isn't possible to disable or enable optimizations for each prover separately.

Finally, Asm3 has no knowledge of WhyML definitions which results in several issues during VC generation. For example it cannot resolve identifiers that are defined in WhyML code[19]. Another example of an issue that occurs is that Asm3 doesn't know the radix of range types defined in a WhyML file. Because of this variables of these types cannot be used in combination with the `::` operator.

While Asm3 can be useful despite these issues, these issues could be mitigated by integrating part of Asm3's features in Why3 itself. Ideally Why3 should be extended in such a way that a syntax level translation from Asm3 to WhyML would be possible without losing functionality. This would mean adding the following features to Why3:

- Recording the flow of information in the generated VCs and adding a transformation to discard premises based on this information. Recording what variable incarnations were produced by an effect. This doesn't require any changes to the WhyML language and has use outside of modeling assembly code. Why3 already includes some information such as premise names as annotations in its formulas, so including this information wouldn't require large changes to the structure of VCs.

[18] Version 1.4 and 1.7 of CVC4 were both used instead of just CVC4 1.4 that was used in this article. The reason for this is that CVC4 1.7 was needed to verify a goal that required working with bitvector theories.

[19] For this reason Asm3 requires the user to specify a namespace in front of identifiers it itself cannot resolve.

- Adding a transformation to model type invariants with predicates. The issue with modeling type invariants shouldn't be much of a problem (modeling without abstract types) as it has worked in the past. Moreover this wouldn't necessarily require changes to the VC generator: it could be implemented as a transformation to Why3's current VCs.
- Adding index references. I personally do not expect the maintainers of Why3 to accept something similar to the index references found in Asm3 as the aliasing limitations of Why3's own references appear to be intentional and does not allow the user to model anything that couldn't be modeled using maps. The least invasive way to support something similar in WhyML would be to include a modified `may_alias` annotation and a more fine grained `writes` annotation for mutable maps. For example, for aliasing the following annotation could be introduced: `may_alias { x with y in m }` where `m` is a mutable map and `x` and `y` can be indices into this map. The `writes` annotation could be extended by specifying the modified map indices, for example: `writes { m[x;y] }` instead of just `writes { m }`. If this information is then retained in the generated VCs a transformation could be added to compute an updated VC where for incarnations m_i of `m`, occurrences of $m_i[x]$ could be replaced with some t if the system can compute that $m_i[x] = t$.

Aside from helping resolve VC related issues a number of features would become easier to support if the Asm3 plugin would simply transform its Asm3 input to an equivalent WhyML AST representation. For example, it would become easier to fix the following shortcomings:

- Ghost code is currently missing from the Asm3 language.
- While it is possible to import WhyML definitions in Asm3 code the reverse isn't possible.

7.2 Missing features

Asm3 currently lacks several important features.

7.2.1 Jumps

It currently isn't possible to model jumps in Asm3. An approach to modeling assembly code containing branches with Why3 has been described in [11], something similar could be implemented for Asm3. Appendix D contains an outline for an alternative approach to extend Asm3 with support for branching. However I have not conducted sufficient research on if this approach would indeed work well enough or how this approach would compare to the aforementioned article or other published approaches.

7.2.2 Code extraction

Another important feature that is missing is code extraction. It wouldn't take much work to export a list of instructions with arguments along with labels. This output wouldn't match the syntax of expected by an assembler, however. Some scripting would be needed to transform it to assembly code of the desired assembly language, though it should be possible to perform this step with a number of regular-expression replace rules.

7.3 Further improvements

7.3.1 Improved memory separation

While index references work well for modeling things like registers, Asm3 brings little improvement for modeling memory such as the stack or heap. During my work [13]

on verifying an implementation of curve25519 written in AVR assembly I found that proving that the verification times of assertions stating which areas of main memory aren't modified rise sharply as the number of store instruction increases. Being able to separate accessed memory from the rest of the memory would help here. It should be possible to generalize index references to the most common patterns of accessing main memory.

Somewhat related, when working with values constructed out of multiple registers or multiple bytes of main memory it currently isn't straightforward to work with just this value and abstract away from the underlying memory. It could be useful if the system could automatically detect if just composite values are referenced instead of one of the underlying values. The system could then optimize the underlying values away. For example, if an assembly routine works with 256 bits wide values it may often be useful to abstract away from operations on individual bytes or registers and only reason about the composite values.

7.3.2 Improving effect selection

While the current system of automatically inferring relations between goals and premises based on writes and reads works reasonably well, it would be useful to be able to manually specify on which variables a postcondition or assertion provides information.

8 Related work

The Vale DSL introduced in [3] is a DSL for writing efficient verified assembly code. This language is somewhat similar to the Asm3 language but has a greater emphasis on writing new verified code instead of verifying existing implementations. Unlike Asm3 this language can generate varying assembly code based on conditions checked during the code extraction phase, making Vale more flexible. Another large advantage of Vale is that its VC generator is not part of the trusted code base. Aside from the different focus Asm3 has some advantages over Vale. While Vale/F* introduced in [9] generates more efficient VCs than the first version its VCs still seem significantly less efficient than the VCs generated by Asm3[20]. In my personal experience Vale/F* features a much steeper learning curve than working with Why3[21] and Asm3. For example, a user of Vale/F* needs to be able to understand both the Vale and F* languages, which are not alike. Adding support for another instruction set architecture requires writing a significant amount of code in both Vale and F* while it takes relatively little effort to model a useful subset of a new architecture in Asm3. Finally, Why3 allows for the use of a wide variety of provers while F* currently depends on Z3.

Pereira and Sousa describe a tool, `army`, that generates purely sequential Why3 programs for annotated ARM assembly code [11]. These generated programs are related to the original code in that, if these generated programs are proven correct the initial assembly program is also correct. The Why3 programs generated by this tool appear to be similar to the Why3 programs written to model AVR assembly programs in [12, 13], on which this thesis is based. The focus of that paper is different from the latter papers as well as this thesis. Pereira and Sousa's paper focuses on being able to verify unstructured programs, with an emphasis on verifying the complexity of these programs. Marc's work in [12] on the other hand focuses on methods of letting Why3 generate more efficient verification conditions by breaking assembly routines up with abstract blocks and by using ghost code. This thesis iterates on this by allowing code and proof reuse by using macros

[20] I cannot fully exclude the possibility that this is the result of the vast difference in experience I have in working with WhyML or Asm3 and Vale.

[21] Why3 is reported to be a suitable tool for performing deductive program verification, for software engineers with no prior experience in this field [14].

Kamkin et al. [10] suggest an approach to verify both ACSL annotated C code and the assembly code generated for this code. The goal of Asm3 is a bit different as it is made with a focus on handwritten assembly code. It is also not clear if the approach from Kamkin et al. scales to larger C methods. The C method resulting the largest number of instructions still resulted in just 32 instructions. The instructions of the RISC-V instruction set architecture used in this article generally have only one effect instead of several effects. Because of this the effect selection optimization described in this thesis would be somewhat less useful for the case described in the case study.

9 Conclusion

I have presented the Asm3 DSL for modeling assembly code in Why3 and implemented support for this language in a Why3 plugin containing its own VC generator. Compared to Why3's WhyML language, this language allows for more succinct specifications for code working with statically addressed memory, such as registers. For the 64×64 bits Karatsuba multiplication routine, the optimizations the Asm3 plugin can apply during VC generation resulted in increased prover performance. Using these optimizations led to a significant increase in the number of goals that could be verified by multiple provers, as well as a decrease in verification times. The optimizations also seem to greatly improve how verification time scales with the number of verified instructions. Of the provers tested, the Z3 and Alt-Ergo provers were most affected by the optimizations. As the Asm3 plugin doesn't contain any optimizations that are specifically geared towards verifying code containing multiplications, I expect these findings to hold up in general. Though the presented language and plugin currently lack too many features to be useful in most real-world applications, the language features and VC generation optimizations should be useful in many applications. While this thesis has focused on verifying the specifications of assembly code, the VC generation optimizations could also be applied when verifying code written in high-level languages.

The code of the Asm3 plugin can be found at `https://gitlab.science.ru.nl/jmoerman/asm3-why3-plugin`. The commit corresponding to the version presented in this thesis is abc89516314aff363d8eb474e86d8bec53b18fb3.

References

[1] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. Cvc4. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 171–177.

[2] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages* (2011), pp. 53–64.

[3] Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K. R. M., Lorch, J. R., Parno, B., Rane, A., Setty, S., and Thompson, L. Vale: Verifying high-performance cryptographic assembly code. In *26th {USENIX} Security Symposium ({USENIX} Security 17)* (2017), pp. 917–934.

[4] Conchon, S., Coquereau, A., Iguernlala, M., and Mebsout, A. Alt-ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories* (2018).

[5] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[6] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (1975), 453–457.

[7] Filliâtre, J.-C., Gondelman, L., and Paskevich, A. The spirit of ghost code. *Formal Methods in System Design 48*, 3 (2016), 152–174.

[8] Filliâtre, J.-C., and Paskevich, A. Why3–where programs meet provers. In *European Symposium on Programming* (2013), Springer, pp. 125–128.

[9] Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., and Swamy, N. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–30.

[10] Kamkin, A., Khoroshilov, A., Kotsynyak, A., and Putro, P. Deductive binary code verification against source-code-level specifications. In *International Conference on Tests and Proofs* (2020), Springer, pp. 43–58.

[11] Pereira, M., and de Sousa, S. M. Complexity checking of arm programs, by deduction. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (2014), pp. 1309–1314.

[12] Schoolderman, M. Verifying branch-free assembly code in why3. In *Working Conference on Verified Software: Theories, Tools, and Experiments* (2017), Springer, pp. 66–83.

[13] Schoolderman, M., Moerman, J., Smetsers, S., and van Eekelen, M. Efficient verification of optimized code. In *NASA Formal Methods* (Cham, 2021), A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, Eds., Springer International Publishing, pp. 304–321.

[14] Schoolderman, M., Smetsers, S., and van Eekelen, M. Is deductive program verification mature enough to be taught to software engineers? In *Proceedings of the 8th Computer Science Education Research Conference* (2019), pp. 50–57.

[15] Schulz, S., Cruanes, S., and Vukmirović, P. Faster, higher, stronger: E 2.3. In *Proc. of the 27th CADE, Natal, Brasil* (2019), P. Fontaine, Ed., no. 11716 in LNAI, Springer, pp. 495–507.

Appendix

A Instruction definitions used in section 5

Listing 31: Definitions for flags in special register SREG

```
type bit = <range 0 1>
weak type rb_index = <range 0 8> (* To constrain indexes into SREG *)

(* variables (x: srbit) implicitly index mutable variable sreg *)
index srbit, sreg : rb_index, bit

(* Carry flag *)
let constant cf: srbit = 0
(* Zero flag *)
let constant zf: srbit = 1
```

```
(* Negative Flag *)
let constant nf: srbit = 2
(* Two's complement overflow indicator *)
let constant vf: srbit = 3
(* N ⊕ V, for signed tests *)
let constant sf: srbit = 4
(* Half Carry Flag *)
let constant hf: srbit = 5
(* Transfer bit used by BLD and BST instructions *)
let constant tf: srbit = 6
(* Global Interrupt Enable/Disable Flag *)
let constant ifl: srbit = 7
```

Listing 32: Axiomatic definition of *Add without Carry* instruction

```
val instr add (rd rr: reg)
writes { rd, cf, hf, sf, vf, nf, zf }
reads { rd, rr }
ensures Post_rd { $rd = mod (old ($rd + $rr)) 256 }
ensures Post_cf { $cf = div (old ($rd + $rr)) 256 }
ensures Post_hf {
  $hf = div (
    old ((mod ($rd) 16) + (mod ($rr) 16))
  ) 16
}
ensures Post_sf { $sf = mod ($vf + $nf) 2 }
ensures Post_vf {
  let rd7 = old ($rd) >= 128 in
  let rr7 = old ($rr) >= 128 in
  let r7 = mod (old ($rd + $rr)) 256 >= 128 in
  $vf = if (
        (rd7 /\ rr7 /\ not r7) \/
        ((not rd7) /\ (not rr7) /\ r7)
  ) then 1 else 0
}
ensures Post_nf { $nf = div (mod (old ($rd + $rr)) 256) 0x80 }
ensures Post_zf {
  $zf = if mod (old ($rd + $rr)) 256 = 0 then 1 else 0
}
may_alias { rd with rr }
```

Listing 33: Axiomatic definition of *Add with Carry* instruction

```
val instr adc (rd rr: reg)
writes { rd, cf, hf, sf, vf, nf, zf }
reads { rd, rr }
ensures Post_rd { $rd = mod (old ($rd + $rr + $cf)) 256 }
ensures Post_cf { $cf = div (old ($rd + $rr + $cf)) 256 }
ensures Post_hf {
  $hf = div (
    old ((mod ($rd) 16) + (mod ($rr) 16) + $cf)
  ) 16
}
ensures Post_sf { $sf = mod ($vf + $nf) 2 }
ensures Post_vf {
  let rd7 = old ($rd) >= 128 in
  let rr7 = old ($rr) >= 128 in
  let r7 = mod (old ($rd + $rr + $cf)) 256 >= 128 in
  $vf = if (
        (rd7 /\ rr7 /\ not r7) \/
        ((not rd7) /\ (not rr7) /\ r7)
  ) then 1 else 0
}
```

```
ensures Post_nf { $nf = div (mod (old ($rd + $rr + $cf)) 256) 0x80 }
ensures Post_zf {
  $zf = if mod (old ($rd + $rr + $cf)) 256 = 0 then 1 else 0
}
may_alias { rd with rr }
```

Listing 34: Axiomatic definition of *Multiply Unsigned* instruction

```
val instr mul (rd rr: reg)
writes { r0, r1, cf, zf }
reads { rd, rr }
ensures Post_r0 { $r0 = mod (old ($rd * $rr)) 256 }
ensures Post_r1 { $r1 = div (old ($rd * $rr)) 256 }
ensures Post_cf { $cf = div (old ($rd * $rr)) 0x8000 }
ensures Post_zf { $zf = if old ($rd * $rr) = 0 then 1 else 0 }
may_alias { r0, r1 with rd, rr }
```

Listing 35: Axiomatic definition of the *Clear Register* instruction

```
val instr clr (rd: reg)
writes { rd }
writes { sf, vf, nf, zf }
ensures Post_rd { $rd = 0 }
ensures Post_sf { $sf = 0 }
ensures Post_vf { $vf = 0 }
ensures Post_nf { $nf = 0 }
ensures Post_zf { $zf = 1 }
```

B Observations made during the process of verifying the 64×64 karatsuba multiplication routine

The default SMT2 drivers used to generate the input for the CVC4 and Z3 provers, often performed poorly. Using the alternative "noBV" (no bitvector) driver[22] for CVC4 1.4 often yielded much better results. This difference appears to be mainly caused by the way the `mod` and `div` functions are modeled in the resulting SMT2 prover input. When the default drivers are used for CVC4 and Z3 built in definitions for these functions are used while this isn't the case when this driver is used.

This driver was needed to get CVC4 and Z3 to verify the proof goals generated for the preconditions of the load and store instructions. When verifying the proof goals generated for the assertions and postconditions the default drivers for both versions of Z3 performed significantly better. For these goals the default drivers of both versions of CVC4 performed similarly to the CVC4 noBV driver. Using that driver some proof goals could be verified that couldn't be verified with the default drivers, but the reverse was also true.

C Asm3 code for the 64×64 Karatsuba multiplication routine

Code listing starts next page.

[22] The driver file for this alternative is `cvc4.drv`.

```
module KaratAvr

use_why3 int.Int
use_why3 int.EuclideanDivision
use_why3 map.Map
use_why3 bv.BV8
use_why3 bv.Pow2int
use_why3 int.Abs

(* Lemmas for instructions modeled with bitvectors *)
lemma of_int_zeros: of_int 0 = zeros
lemma of_int_ones: of_int 0xFF = ones

lemma xor_0: forall w. 0 <= w < 256 -> t'int (bw_xor (of_int w) zeros) = w
lemma xor_1: forall w. 0 <= w < 256 -> t'int (bw_xor (of_int w) ones) = 255-w

lemma or_0: forall w. bw_or zeros w = w

(* Lemmas for multiplications *)
lemma mul_bound_preserve:
  forall x y l. 0 <= x <= l -> 0 <= y <= l -> x*y <= l*l

(* necessary for goals Compute_L, Add_H_to_L_high1 and M_computation *)
lemma byte_mult_range:
  forall i j : int. -1 < i < 256 /\ -1 < j < 256 -> -1 < i*j <= 255*255

(** Program definitions ************************************************)

weak_type byte = <range 0 255>
weak_type bit = <range 0 1>
weak_type rb_index = <range 0 7> (* To constrain indexes into SREG *)
weak_type reg_index = <range 0 31> (* To constrain the range of registers *)

index reg = create_index regs (reg_index, byte)
index srbit = create_index sreg (rb_index, bit)

(** Register definitions ***********************************************)
let constant rX: reg = 26
let constant rY: reg = 28
let constant rZ: reg = 30

let constant r0: reg = 0
let constant r1: reg = 1
let constant r2: reg = 2
let constant r3: reg = 3
let constant r4: reg = 4
let constant r5: reg = 5
let constant r6: reg = 6
let constant r7: reg = 7
let constant r8: reg = 8
let constant r9: reg = 9
let constant r10: reg = 10
let constant r11: reg = 11
let constant r12: reg = 12
let constant r13: reg = 13
let constant r14: reg = 14
let constant r15: reg = 15
let constant r16: reg = 16
let constant r17: reg = 17
let constant r18: reg = 18
let constant r19: reg = 19
let constant r20: reg = 20
let constant r21: reg = 21
let constant r22: reg = 22
let constant r23: reg = 23
let constant r24: reg = 24
let constant r25: reg = 25
let constant r26: reg = 26
let constant r27: reg = 27
let constant r28: reg = 28
let constant r29: reg = 29
```

```
72  let constant r30: reg = 30
73  let constant r31: reg = 31
74
75  (** SREG definitions ******************************************************)
76  (* carry flag *)
77  let constant cf: srbit = 0
78  (* T flag *)
79  let constant tf: srbit = 6
80  (* other flags are not modelled *)
81
82  (* MOV - Copy Register *)
83  val instr mov (rd rr: reg)
84  writes { rd }
85  reads { rr }
86  ensures { $rd = old ($rr) }
87  may_alias { rd with rr }
88
89  (* MOVW - Copy Register Word *)
90  val instr movw (rd rr: reg)
91  writes { rd, rd+1 }
92  reads { rr, rr+1 }
93  requires { rd < 31 /\ rr < 31 }
94  ensures { $rd = old ($rr) }
95  ensures { $(rd+1) = old $(rr+1) }
96  may_alias { rd with rr,rr+1 }
97  may_alias { rd+1 with rr,rr+1 }
98
99  (* MUL- Multiply Unsigned *)
100 val instr mul (rd rr: reg)
101 writes { r0, r1, cf }
102 reads { rd, rr }
103 ensures { $r0 = EuclideanDivision.mod (old ($rd * $rr)) 256 }
104 ensures { $r1 = EuclideanDivision.div (old ($rd * $rr)) 256 }
105 may_alias { r0 with rd,rr }
106 may_alias { r1 with rd,rr }
107
108 (* ADD - Add without Carry *)
109 val instr add (rd rr: reg)
110 writes { rd, cf }
111 reads { rd, rr }
112 ensures { $rd = EuclideanDivision.mod (old ($rd + $rr)) 256 }
113 ensures { $cf = EuclideanDivision.div (old ($rd + $rr)) 256 }
114 may_alias { rd with rr }
115
116 (* ADC - Add with Carry *)
117 val instr adc (rd rr: reg)
118 writes { rd, cf }
119 reads { rd, rr, cf }
120 ensures { $rd = EuclideanDivision.mod (old ($rd + $rr + $cf)) 256 }
121 ensures { $cf = EuclideanDivision.div (old ($rd + $rr + $cf)) 256 }
122 may_alias { rd with rr }
123
124 (* SUB- Subtract without Carry *)
125 val instr sub (rd rr: reg)
126 writes { rd, cf }
127 reads { rd, rr }
128 ensures { $rd = EuclideanDivision.mod (old ($rd - $rr)) 256 }
129 ensures { $cf = - EuclideanDivision.div (old ($rd - $rr)) 256 }
130 may_alias { rd with rr }
131
132 (* SBC- Subtract with Carry *)
133 val instr sbc (rd rr: reg)
134 writes { rd, cf }
135 reads { rd, rr, cf }
136 ensures { $rd = EuclideanDivision.mod (old ($rd - $rr - $cf)) 256 }
137 ensures { $cf = - EuclideanDivision.div (old ($rd - $rr - $cf)) 256 }
138 may_alias { rd with rr }
139
140 (* DEC- Decrement *)
141 val instr dec (rd: reg)
142   writes { rd }
```

```
143     ensures { $rd = EuclideanDivision.mod (old ($rd) - 1) 256 }
144
145   (* CLR - Clear Register *)
146   val instr clr (rd: reg)
147   writes { rd }
148   ensures { $rd = 0 }
149
150   (* EOR - Exclusive OR *)
151   val instr eor (rd: reg) (rr: reg)
152     writes { rd }
153     reads { rd, rr }
154     ensures { $rd = old (BV8.t'int (BV8.bw_xor (BV8.of_int ($rd)) (BV8.of_int
    ($rr)))) }
155
156   (* ASR - Arithmetic Shift Right *)
157   val instr asr (rd: reg)
158     writes { rd, cf }
159     ensures { let rdv = old ($rd) in
160               $rd = EuclideanDivision.div rdv 2 + 128 * EuclideanDivision.div
    rdv 128 }
161     ensures { $cf = EuclideanDivision.mod (old ($rd)) 2 }
162
163   (* BST - Bit Store from Bit in Register to T Flag in SREG *)
164   val instr bst (rd: reg) (bit: rb_index)
165     writes { tf }
166     reads { rd }
167     ensures { $tf = if BV8.nth (BV8.of_int ($rd)) bit then 1 else 0 }
168
169   function bitset (w: BV8.t) (b: int) (v: bool): BV8.t =
170     let mask = BV8.lsl (BV8.of_int 1) b in
171     if v then
172       BV8.bw_or w mask
173     else
174       BV8.bw_and w (BV8.bw_not mask)
175
176   (* BLD - Bit Load from the T Flag in SREG to a Bit in Register. *)
177   val instr bld (rd: reg) (bit: rb_index)
178     writes { rd }
179     ensures { $rd = BV8.t'int (bitset (BV8.of_int (old $rd)) bit (if ($tf) = 1
    then true else false)) }
180
181   weak_type ram = map int byte
182
183   mutable val mem: ram
184
185   (* ST (STD) – Store Indirect From Register to Data Space using Index Y *)
186   val instr std (dst: reg) (ofs: reg_index) (rr: reg)
187     writes { mem }
188     reads { dst, dst+1, rr }
189     requires { 32 <= ($dst...2) + ofs < Pow2int.pow2 16 }
190     ensures { $mem = old ($mem)[($dst...2)+ofs <- $rr] }
191
192   (* LD - Load Indirect from data space to Register using Index X
193    *** LD Rd, X+ ***)
194   val instr ld_inc (dst src: reg)
195     writes { dst, src, src+1 }
196     reads { mem, src, src+1 }
197     requires { 32 <= $src...2 < Pow2int.pow2 16-1 }
198     ensures { $dst = old (($mem)[$src...2]) }
199     ensures { $src...2 = old ($src...2) + 1 }
200
201   (* LD (LDD)- Load Indirect from data space to Register using Index Y
202    *** LDD Rd, Y+q ***)
203   val instr ldd (dst: reg) (src: reg) (q: reg_index)
204     writes { dst }
205     reads { mem, src, src+1 }
206     requires { 32 <= $src...2 + q < Pow2int.pow2 16 }
207     ensures { $dst = old (($mem)[$src...2 + q]) }
208
209   let entry["karatsuba64.asm"] karatsuba64
210     requires { 32 <= $rX...2 < 32768 }
```

```
211     requires { 32 <= $rY...2 < 32768 }
212     requires { 32 <= $rZ...2 < 32768 }
213     writes { mem, cf, tf, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11,
    r12, r13, r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, rX,
    rX+1, rY, rY+1 }
214     ensures Result_at_Z {
215       let i = $rZ...2 in
216       let j = old ($rX...2) in
217       let k = old ($rY...2) in
218       ($mem)[i]...16 = old ((($mem)[j]...8) * (($mem)[k]...8))
219     }
220     ensures Unchanged_memory {
221       let i = $rZ...2 in
222       forall x. x < i \/ x > i + 15 -> ($mem)[x] = old ($mem)[x]
223     }
224   =
225    (* init rZero registers *)
226     clr r20;
227     clr r21;
228     movw r16 r20;
229
230    (* read arguments A and B from SRAM *)
231     ld_inc r2 rX;
232     ld_inc r3 rX;
233     ld_inc r4 rX;
234     ld_inc r5 rX;
235     ldd r6 rY r0;
236     ldd r7 rY r1;
237     ldd r8 rY r2;
238     ldd r9 rY r3;
239
240   label L00 in
241   begin
242   ensures Compute_L { $r10...8 = $r2...4 * $r6...4 }
243     mul r2 r8; (* a0*b2 *)
244     movw r12 r0;
245     mul r2 r6; (* a0*b0 *)
246     movw r10 r0;
247     mul r2 r7; (* a0*b1 *)
248     add r11 r0;
249     adc r12 r1;
250     adc r13 r21;
251     mul r3 r9; (* a1*b3 *)
252     movw r14 r0;
253
254     mul r2 r9; (* a0*b3 *)
255     movw r18 r0;
256     mul r3 r6; (* a1*b0 *)
257     add r11 r0;
258     adc r12 r1;
259     adc r13 r18;
260     adc r19 r21;
261     mul r3 r7; (* a1*b1 *)
262     add r12 r0;
263     adc r13 r1;
264     adc r19 r21;
265     mul r4 r9; (* a2*b3 *)
266     add r14 r19;
267     adc r15 r0;
268     adc r16 r1;
269
270     mul r4 r8; (* a2*b2 *)
271     movw r18 r0;
272     mul r4 r6; (* a2*b0 *)
273     add r12 r0;
274     adc r13 r1;
275     adc r14 r18;
276     adc r19 r21;
277     mul r3 r8; (* a1*b2 *)
278     add r13 r0;
279     adc r14 r1;
```

```
    adc r19 r21;
    mul r5 r9; (* a3*b3 *)
    add r15 r19;
    adc r16 r0;
    adc r17 r1;

    mul r5 r7; (* a3*b1 *)
    movw r18 r0;
    mul r4 r7; (* a2*b1 *)
    add r13 r0;
    adc r18 r1;
    adc r19 r21;
    mul r5 r6; (* a3*b0 *)
    add r13 r0;
    adc r18 r1;
    adc r19 r21;
    mul r5 r8; (* a3*b2 *)
    add r14 r18;
    adc r0 r19;
    adc r1 r21;
    add r15 r0;
    adc r16 r1;
    adc r17 r21;
  end;

   (*    load a4..a7 and b4..b7    *)
    ldd r22 rY 4;
    ldd r23 rY 5;
    ldd r24 rY 6;
    ldd r25 rY 7;
    movw r28 r20;
    ld_inc r18 rX;
    ld_inc r19 rX;
    ld_inc r20 rX;
    ld_inc r21 rX;
    movw r26 r28;

    std rZ 0 r10;
    std rZ 1 r11;
    std rZ 2 r12;
    std rZ 3 r13;

  label L11 in
  begin
  ensures M_factors_computation { (* compute the factors |A_l − A_h| and |B_l −
  B_h| *)
    let a = old ($r2...4) in
    let b = old ($r18...4) in
    let c = old ($r6...4) in
    let d = old ($r22...4) in
    (if $tf = 0 then 1 else -1)*($r2...4*$r6...4) = (a - b)*(c - d) (*
  transformations: elimininate_if, split *)
    by
    ($tf = 0 <-> ((a - b) < 0 <-> (c - d) < 0)) /\
    ($r2...4*$r6...4 = Abs.abs ((a - b)*(c - d))
      by ((
        $r2...4 = Abs.abs (a - b) /\
        $r6...4 = Abs.abs (c - d) /\
        (
          Abs.abs (a - b) * Abs.abs (c - d) = Abs.abs ((a - b) * (c - d)) by (
            forall x y. Abs.abs x * Abs.abs y = Abs.abs (x*y)
          )
        )
      )
    ))
    } using neg_mult_is_pos (* transformations: split(_right) until reaching
  sub-goals  *)

  label B in
   (*    subtract a0 a4      *)
    sub r2 r18;
```

```
348     sbc r3 r19;
349     sbc r4 r20;
350     sbc r5 r21;
351    (* 0xff if carry and 0x00 if no carry *)
352     sbc r0 r0;
353
354    (*    subtract b0 b4    *)
355     sub r6 r22;
356     sbc r7 r23;
357     sbc r8 r24;
358     sbc r9 r25;
359    (* 0xff if carry and 0x00 if no carry *)
360     sbc r1 r1;
361     assert { $r0 = if ($r2...4 < $r18...4) at B then 0xFF else 0x00 };
362     assert { $r1 = if ($r6...4 < $r22...4) at B then 0xFF else 0x00 };
363
364    (*    absolute values    *)
365     eor r2 r0;
366     eor r3 r0;
367     eor r4 r0;
368     eor r5 r0;
369     eor r6 r1;
370     eor r7 r1;
371     eor r8 r1;
372     eor r9 r1;
373
374     sub r2 r0;
375     sbc r3 r0;
376     sbc r4 r0;
377     sbc r5 r0;
378     sub r6 r1;
379     sbc r7 r1;
380     sbc r8 r1;
381     sbc r9 r1;
382
383     eor r0 r1;
384     bst r0 0 ;
385     assert { $tf = if (($r0) = 0xFF) then 1 else 0 };
386     assert {
387       let an = ($r2...4 < $r18...4) at B in
388       let bn = ($r6...4 < $r22...4) at B in
389       $tf = if (an \/ bn) /\ not (an /\ bn) then 1 else 0
390     };
391     assert neg_mult_is_pos {forall x y. x < 0 /\ y < 0 -> x*y > 0};
392
393     (* useful as (tf = 0) <-> xv < 0 <-> yv < 0 *)
394     assert tf_abs_property {
395       forall x y.
396         x * y = if (x < 0 <-> y < 0) then
397           Abs.abs (x * y) else
398           - Abs.abs (x * y)
399     }; (* Proving general property in clean proof context *)
400     assert {
401       let xv = ($r2...4 - $r18...4) at B in
402       let yv = ($r6...4 - $r22...4) at B in
403       xv * yv = if (xv < 0 <-> yv < 0) then
404         Abs.abs (xv * yv) else
405         - Abs.abs (xv * yv)
406     } using tf_abs_property;
407   end;
408
409   begin
410   ensures Add_H_to_L_high1 { $r14...4::$r28...2::$r18...2 = old (($r14...4) +
      $r18...4*$r22...4) }
411
412    (*    compute h  (l4 l5 l6 l7)    *)
413     mul r18 r22;
414     add r14 r0;
415     adc r15 r1;
416     adc r16 r26;
417     adc r29 r26;
```

```
418
419     mul r18 r23;
420     add r15 r0;
421     adc r16 r1;
422     adc r29 r26;
423     mul r19 r22;
424     add r15 r0;
425     adc r16 r1;
426     adc r17 r29;
427     adc r28 r26;
428
429     mul r18 r24;
430     add r16 r0 ;
431     adc r17 r1;
432     adc r28 r26;
433     mul r19 r23;
434     add r16 r0;
435     adc r17 r1;
436     adc r28 r26;
437     mul r20 r22;
438     add r16 r0;
439     adc r17 r1;
440     adc r28 r26;
441
442     clr r29;
443     mul r18 r25;
444     add r17 r0;
445     adc r28 r1;
446     adc r29 r26;
447     mul r19 r24;
448     add r17 r0;
449     adc r28 r1;
450     adc r29 r26;
451     mul r20 r23;
452     add r17 r0;
453     adc r28 r1;
454     adc r29 r26;
455     mul r21 r22;
456     add r17 r0;
457     adc r28 r1;
458     adc r29 r26;
459
460     mul r19 r25;
461     movw r18 r26;
462     add r28 r0;
463     adc r29 r1;
464     adc r18 r26;
465     mul r20 r24;
466     add r28 r0;
467     adc r29 r1;
468     adc r18 r26;
469     mul r21 r23;
470     add r28 r0;
471     adc r29 r1;
472     adc r18 r26;
473
474     mul r20 r25;
475     add r29 r0;
476     adc r18 r1;
477     adc r19 r26;
478     mul r21 r24;
479     add r29 r0;
480     adc r18 r1;
481     adc r19 r26;
482
483     mul r21 r25;
484     add r18 r0;
485     adc r19 r1;
486   end;
487
488   begin
```

```
489  ensures M_computation { $r20...6::$r2...2 = old ($r2...4 * $r6...4) }
490
491   (*    compute m    *)
492    mul r2 r6;
493    movw r20 r0;
494
495    movw r22 r26;
496    mul r2 r7;
497    add r21 r0;
498    adc r22 r1;
499    mul r3 r6;
500    add r21 r0;
501    adc r22 r1;
502    adc r23 r26;
503
504    movw r24 r26;
505    mul r2 r8;
506    add r22 r0;
507    adc r23 r1;
508    adc r24 r26;
509    mul r3 r7;
510    add r22 r0;
511    adc r23 r1;
512    adc r24 r26;
513    mul r4 r6;
514    add r22 r0;
515    adc r23 r1;
516    adc r24 r26;
517
518    mul r2 r9;
519    add r23 r0;
520    adc r24 r1;
521    adc r25 r26;
522    mul r3 r8;
523    add r23 r0;
524    adc r24 r1;
525    adc r25 r26;
526    mul r4 r7;
527    add r23 r0;
528    adc r24 r1;
529    adc r25 r26;
530    mul r5 r6;
531    add r23 r0;
532    adc r24 r1;
533    adc r25 r26;
534
535    mul r3 r9;
536    movw r2 r26;
537    add r24 r0;
538    adc r25 r1;
539    adc r2 r27;
540    mul r4 r8;
541    add r24 r0;
542    adc r25 r1;
543    adc r2 r27;
544    mul r5 r7;
545    add r24 r0;
546    adc r25 r1;
547    adc r2 r27;
548
549    mul r4 r9;
550    add r25 r0;
551    adc r2 r1;
552    adc r3 r27;
553    mul r5 r8;
554    add r25 r0;
555    adc r2 r1;
556    adc r3 r27;
557
558    mul r5 r9;
559    add r2 r0;
```

```
560     adc r3 r1;
561
562   end;
563
564   begin
565   ensures Add_H_to_L_high2 { $r10...8::$cf = old($r10...8 +
      $r14...4::$r28...2::$r18...2) }
566    (*    add l4 h0 to l0 and h4    *)
567     add r10 r14;
568     adc r11 r15;
569     adc r12 r16;
570     adc r13 r17;
571     adc r14 r28;
572     adc r15 r29;
573     adc r16 r18;
574     adc r17 r19;
575    (* store carrrY in r26 *)
576   end;
577
578   begin
579   ensures Correct_M_sign { $r20...6::$r2...2 = (if $cf = 1 then 1 else
      0)*(Pow2int.pow2 64 - 1) - ((($r2...4 - $r18...4)*($r6...4 - $r22...4)) at
      L11) } (* rewrite pow2_64, eliminate_if, split, split *)
580   ensures Carry_value { let cor = $r26 + (Pow2int.pow2 8 + Pow2int.pow2 16 +
      Pow2int.pow2 24) * $r0 in cor = (old $cf) - $cf \/ cor = Pow2int.pow2 32 +
      (old $cf) - $cf }
581
582    (*    process sign bit      *)
583   label B in
584     bld r27 0;
585   assert { $r27 = $tf by $r27 at B = 0 };
586     dec r27;
587   assert { $r27 = 0xFF*(1 - $tf) };
588
589    (* merge carry and borrow *)
590     adc r26 r27;
591     mov r0 r26;
592     asr r0;
593
594   label B in
595    (* invert all bits or do nothing *)
596     eor r20 r27;
597     eor r21 r27;
598     eor r22 r27;
599     eor r23 r27;
600     eor r24 r27;
601     eor r25 r27;
602     eor r2  r27;
603     eor r3  r27;
604     assert {$r27 = 0xFF -> (
605       $r3 = 255 - $r3 at B /\
606       $r2 = 255 - $r2 at B /\
607       $r25 = 255 - $r25 at B /\
608       $r24 = 255 - $r24 at B /\
609       $r23 = 255 - $r23 at B /\
610       $r22 = 255 - $r22 at B /\
611       $r21 = 255 - $r21 at B /\
612       $r20 = 255 - $r20 at B
613     )};
614     assert {$r27 = 0 -> (
615       $r3  = $r3  at B /\
616       $r2  = $r2  at B /\
617       $r25 = $r25 at B /\
618       $r24 = $r24 at B /\
619       $r23 = $r23 at B /\
620       $r22 = $r22 at B /\
621       $r21 = $r21 at B /\
622       $r20 = $r20 at B
623     )};
624     add r27 r27; (* sets carry flag if r27 = 0xff *)
625   end;
```

```
626
627 begin
628 ensures Add_M { $r10...8::$r28...2::$r18...2::$cf = old($cf +
    $r10...8::$r28...2::$r18...2 + $r20...6::$r2...2 + Pow2int.pow2 64*($r26 +
    (Pow2int.pow2 8+Pow2int.pow2 16+Pow2int.pow2 24) * $r0)) }
629
630  (* add in m *)
631   adc r10 r20;
632   adc r11 r21;
633   adc r12 r22;
634   adc r13 r23;
635   adc r14 r24;
636   adc r15 r25;
637   adc r16 r2;
638   adc r17 r3;
639
640  (* propagate carry/borrow *)
641   adc r28 r26;
642   adc r29 r0;
643   adc r18 r0;
644   adc r19 r0;
645 end;
646
647 assert L11_product_limits {
648   let a = ($r2...4::$r18...4) at L11 in
649   let b = ($r6...4::$r22...4) at L11 in
650   let c = Pow2int.pow2 64-1 in
651   0 <= a * b <= c*c by (0 <= a <= c /\ 0 <= b <= c)
652 }; (* transformations: rewrite pow2_64, split, split *)
653
654 assert Mult_after_L11 {
655   $r10...4 at L11::$r10...8::$r28...2::$r18...2 =
656   ( $r2...4::$r18...4 * $r6...4::$r22...4 ) at L11
657 };
658
659   std rZ 4 r10;
660   std rZ 5 r11;
661   std rZ 6 r12;
662   std rZ 7 r13;
663   std rZ 8 r14;
664   std rZ 9 r15;
665   std rZ 10 r16;
666   std rZ 11 r17;
667   std rZ 12 r28;
668   std rZ 13 r29;
669   std rZ 14 r18;
670   std rZ 15 r19
671 end
```

D A possible approach for extending Asm3 with support for branching

I propose to extend Asm3 with two additional annotations. One annotation would allow the user to specify how an instruction or macro continues execution. Another annotation should allow the user to specify what happens if a jump from outside the macro to a label inside the macro occurs. If a macro contains a jump to an assembly label present in its body this jump can be modeled. Modeling jumps forwards would be *similar* to modeling conditional statements, while modeling jumps backwards would be similar to modeling loops. The biggest difference with normal conditional statements and loops is that the branches and loop bodies might themselves contain jumps, as well as labels that can be jumped to.

To generate provable VCs for code containing jumps some additional annotations would be required. If the code jumps forwards no additional annotations would be required. However, if the code jumps to an earlier instruction the user would need to provide a loop variant and invariant. In this case the user would also need to specify on which conditions it would jump to a later instruction, should this be possible. An example of how something like this might look is given in listing 36. This listing demonstrates how a user might annotate code containing jumps and labels that can be jumped to, however, it isn't a good example of how real code containing jumps might look like. Listing 37 gives a more realistic example how real code modeled in Asm3 could look like. Proving the specification of this code would be very similar to proving the specification of the WhyML code from listing 38.

Listing 36: Rough example of how jumps could be implemented in a future version of Asm3

```
let label label1 = "A"
let label label2 = "B"

let macro jump_example1 ( label3 : label )
reads { ··· }
writes { ··· }
(* Pre- and postconditions for execution starting anywhere within
    jump_example1 *)
requires[_] { $r1 = 0 }
ensures[_] { ··· }
ensures[_] { old $5 <> 37 -> $r18 = old $r18 - 1 }
(* after jump_example1 finishes, execution continues at label3 if
 * the value stored in r5 is equal to 37,
 * or at label1 if the value stored in r18 is not 0. *)
jumps_to[label1] { $r5 <> 37 /\ $r18 > 0 }
jumps_to[label3] { $r5 = 37 }
= asm_label label2 in (* inserts the label label2 in front of the next
    instruction *)
  ··· (* Some useful operations *)
  cpi r5 37;
  breq label3; (* conditional jump to label3 *)
  dec r18;
  cp r18 r1;
  brne label1

let macro jump_example2 ( label3 : label )
reads { ··· }
writes { ··· }
(* Specification for execution starting at the start of this macro *)
requires { ··· }
ensures { ··· }
(* Specification for execution starting at label1 *)
```

```
requires[label1] { ··· }
ensures[label1] { ··· }
(* Specification for execution starting at label2 (inside jump_example1) *)
requires[label2] { ··· }
ensures[label2] { ··· }
(* Specification for execution starting at the first instruction or at label1
    *)
ensures[(), label1] { ··· }
(* jump_example2 can finish by jumping to a label not inside its body *)
jumps_to[label3] { ··· }
= ···
  asm_label label1 in
  ···
  jump_example1 label3 with
    | label1 (* the jump to earlier label1 has the following loop (in)variants
     *)
      variant { $r18 }
      invariant { ··· };
  ···
```

Listing 37: An example of how an AVR assembly implementation of 8 bits fibonacci might look if modeled in Asm3 extended with support for jumps.

```
let macro avr_fib (n current : reg) (prev temp i : reg) (loop next_l : label)
requires { fib $n < 256 }
ensures { $current = fib $n }
jumps_to[next_l] { true }
=
  clr current;
  cpi n 0;
  breq next_l;
  clr prev;
  inc current;
  ldi i 2;
  cp i n;
  brsh next_l;
  asm_label loop in
  mov temp current;
  add current prev;
  mov prev temp;
  inc i;
  cp i n;
  brlo loop with
    | loop
      variant { 255 - $i }
      invariant { $n >= $i - 1 /\ $i > 2 }
      invariant { $current = fib ($i - 1) }
      invariant { $prev = fib ($i - 2) };
  jmp next_l
```

Listing 38: WhyML code for which roughly the same proof goals would be generated as for the Asm3 code from listing 37.

```
let byte_ref_fib (n: byte) (current prev temp i: ref byte) : unit
requires { fib n < 256 }
ensures { !current = fib n }
=
  current := (0: byte); (* clr current *)
  if not to_int n = 0 then begin (* cpi current 0; breq next_l *)
    prev := (0: byte); (* clr prev *)
    current := !current + (1: byte); (* inc current *)
```

```
    i := (2: byte); (* ldi i 2 *)
    if not to_int n < to_int !i then begin (* cpi i n; brsh next_l *)
      temp := !current; (* mov temp current *)
      current := !current + !prev; (* add current prev *)
      prev := !temp; (* mov prev temp *)
      i := !i + (1: byte); (* inc i *)
      while to_int n >= to_int !i do (* cpi i n; brlo loop *)
        variant { 255 - !i }
        invariant { n >= !i - 1 /\ !i > 2 }
        invariant { !current = fib (!i - 1) }
        invariant { !prev = fib (!i - 2) }
        temp := !current; (* mov temp current *)
        current := !current + !prev; (* add current prev *)
        prev := !temp; (* mov prev temp *)
        i := !i + (1: byte); (* inc i *)
      done
    end
  end
```