On the correctness of an assembler for the Intel MCS-51 microprocessor*

Dominic P. Mulligan and Claudio Sacerdoti Coen

Dipartimento di Scienze dell'Informazione, Universitá di Bologna

Abstract. We consider the formalisation of an assembler for Intel MCS-51 assembly language in the Matita proof assistant. This formalisation forms a major component of the EU-funded CerCo project, concerning the construction and formalisation of a concrete complexity preserving compiler for a large subset of the C programming language.
The efficient expansion of pseudoinstructions—particularly jumps—into MCS-51 machine instructions is complex. We employ a strategy, involving the use of 'policies', that separates the decision making over how jumps should be expanded from the expansion process itself. This makes the proof of correctness for the assembler significantly more straightforward. We prove, under the assumption of the existence of a correct policy, that the assembly process never fails and preserves the semantics of a subset of assembly programs. Correct policies fail to exist only in a limited number of pathological circumstances. Our assembler is complete with respect to the choice of policy.
Surprisingly, we observe that it is impossible for an optimising assembler to preserve the semantics of every assembly program.

1 Introduction

We consider the formalisation of an assembler for the Intel MCS-51 8-bit microprocessor in the Matita proof assistant [2]. This formalisation forms a major component of the EU-funded CerCo project [4], concerning the construction and formalisation of a concrete complexity preserving compiler for a large subset of the C programming language.

The MCS-51 dates from the early 1980s and is commonly called the 8051/8052. Despite the microprocessor's age, derivatives are still widely manufactured by a number of semiconductor foundries. As a result the processor is widely used, especially in embedded systems development, where well-tested, cheap, predictable microprocessors find their niche.

The MCS-51 has a relative paucity of features compared to its more modern brethren. In particular, the MCS-51 does not possess a cache or any instruction pipelining that would make predicting the concrete cost of executing a single instruction an involved process. Instead, each semiconductor foundry that produces

* The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881

an MCS-51 derivative is able to provide accurate timing information in clock cycles for each instruction in their derivative's instruction set. It is important to stress that this timing information, unlike in more sophisticated processors, is not an estimate, it is a 'definition'. For the MCS-51, if a manufacturer states that a particular opcode takes three clock cycles to execute, then that opcode *always* takes three clock cycles to execute.

This predictability of timing information is especially attractive to the CerCo consortium. We are in the process of constructing a certified, concrete complexity compiler for a realistic processor, and not for building and formalising the worst case execution time tools (WCET—see [3], amongst many others, for an application of WCET technology to microprocessors with more complex designs) that would be necessary to achieve the same result with, for example, a modern ARM or PowerPC microprocessor.

As in most things, what one hand giveth, the other taketh away: the MCS-51's paucity of features, though an advantage in many respects, also quickly become a hindrance, and successfully compiling high-level code for this architecture is a cumbrous and involved process. In particular, the MCS-51 features a relatively miniscule series of memory spaces (including read-only code memory, stack and internal/external random access memory) by modern standards. As a result our C compiler, to have any sort of hope of successfully compiling realistic programs for embedded devices, ought to produce 'tight' machine code. This is not simple and requires the use of optimisations.

For example, the MCS-51 features three unconditional jump instructions: `LJMP` and `SJMP`—'long jump' and 'short jump' respectively—and an 11-bit oddity of the MCS-51, `AJMP`. Each of these three instructions expects arguments in different sizes and behaves in markedly different ways: `SJMP` may only perform a 'local jump'; `LJMP` may jump to any address in the MCS-51's memory space and `AJMP` may jump to any address in the current memory page. Consequently, the size of each opcode is different, and to squeeze as much code as possible into the MCS-51's limited code memory, the smallest possible opcode that will suffice should be selected.

The prototype CerCo C compiler does not attempt to select the smallest jump opcode in this manner, as this was thought to unnecessarily complicate the compilation chain, making the eventual translation and formalisation of the compiler into Matita much harder. Instead, the compiler targets a bespoke assembly language, similar to 'real world' assembly languages, complete with pseudoinstructions including `Jmp` and `Call` instructions. Labels, conditional jumps to labels, a program preamble containing global data and a `MOV` instruction for moving this global data into the MCS-51's one 16-bit register also feature. This latter feature will ease any later consideration of separate compilation in the CerCo compiler. An assembler is used to expand pseudoinstructions into MCS-51 machine code.

However, this assembly process is not trivial, for numerous reasons. For example, our conditional jumps to labels behave differently from their machine code counterparts. At the machine code level, all conditional jumps are 'short',

limiting their range. However, at the assembly level, conditional jumps may jump to a label that appears anywhere in the program, significantly liberalising the use of conditional jumps and further simplifying the design of the CerCo compiler.

Further, trying to naïvely relate assembly programs with their machine code counterparts simply does not work. Machine code programs that fetch from constant addresses in code memory or programs that combine the program counter with constant shifts do not make sense at the assembly level, since the position of instructions in code memory will be known only after assembly and optimisation. More generally, memory addresses can only be compared with other memory addresses. However, checking that memory addresses are only compared against each other at the assembly level is in fact undecidable. In short, we come to the shocking[1] realisation that, with optimisations, the full preservation of the semantics of all assembly programs is impossible. We believe that this revelation is significant for large formalisation projects that assume the existence of a correct assembler. Projects in this class include both the recent CompCert [5,10] and seL4 formalisations [6,7].

Yet, the situation is even more complex than having to expand pseudoinstructions correctly. In particular, when formalising the assembler, we must make sure that the assembly process does not change the timing characteristics of an assembly program for two reasons.

First, the semantics of some functions of the MCS-51, notably I/O, depend on the microprocessor's clock. Changing how long a particular program takes to execute can affect the semantics of a program. This is undesirable.

Second, as mentioned, the CerCo consortium is in the business of constructing a verified compiler for the C programming language. However, unlike CompCert [5,10]—which currently represents the state of the art for 'industrial grade' verified compilers—CerCo considers not just the *extensional correctness* of the compiler, but also its *intensional correctness*. That is, CompCert focusses solely on the preservation of the *meaning* of a program during the compilation process, guaranteeing that the program's meaning does not change as it is gradually transformed into assembly code. However in any realistic compiler (even the CompCert compiler!) there is no guarantee that the program's time properties are preserved during the compilation process; a compiler's 'optimisations' could, in theory, even conspire to degrade the concrete complexity of certain classes of programs. CerCo aims to expand the current state of the art by producing a compiler where this temporal degradation is guaranteed not to happen. Moreover, CerCo's approach lifts a program's timing information to the source (C language) level. This has the advantage of allowing a programmer to reason about a program's intensional properties directly on the source code that they write, not on the code that the compiler produces.

In order to achieve this, CerCo imposes a cost model on programs or, more specifically, on simple blocks of instructions. This cost model is induced by the compilation process itself, and its non-compositional nature allows us to assign different costs to identical blocks of instructions depending on how they are

[1] For us, anyway.

compiled. In short, we aim to obtain a very precise costing for a program by embracing the compilation process, not ignoring it. This, however, complicates the proof of correctness for the compiler proper. In each translation pass from intermediate language to intermediate language, we must prove that both the meaning and concrete complexity characteristics of the program are preserved. This also applies for the translation from assembly language to machine code.

Naturally, this raises a question: how do we assign an *accurate* cost to a pseudoinstruction? As mentioned, conditional jumps at the assembly level can jump to a label appearing anywhere in the program. However, at the machine code level, conditional jumps are limited to jumping 'locally', using a measly byte offset. To translate a jump to a label, a single conditional jump pseudoinstruction may be translated into a block of three real instructions as follows (here, `JZ` is 'jump if accumulator is zero'):

```
        JZ  label                          JZ    size of SJMP instruction
        ...            translates to       SJMP  size of LJMP instruction
label:  MOV A B           ⟹               LJMP  address of label
                                           ...
                                           MOV   A B
```

Here, if `JZ` fails, we fall through to the `SJMP` which jumps over the `LJMP`. Naturally, if `label` is close enough, a conditional jump pseudoinstruction is mapped directly to a conditional jump machine instruction; the above translation only applies if `label` is not sufficiently local. We address the calculation of whether a label is indeed 'close enough' for the simpler translation to be used below.

Crucially, the above translation demonstrates the difficulty in predicting how many clock cycles a pseudoinstruction will take to execute. A conditional jump may be mapped to a single machine instruction or a block of three. Perhaps more insidious is the realisation that the number of cycles needed to execute the instructions in the two branches of a translated conditional jump may be different. Depending on the particular MCS-51 derivative at hand, an `SJMP` could in theory take a different number of clock cycles to execute than an `LJMP`. These issues must also be dealt with in order to prove that the translation pass preserves the concrete complexity of assembly code, and that the semantics of a program using the MCS-51's I/O facilities does not change. We address this problem by parameterising the semantics over a cost model. We prove the preservation of concrete complexity only for the program-dependent cost model induced by the optimisation.

Yet one more question remains: how do we decide whether to expand a jump into an `SJMP` or an `LJMP`? To understand, again, why this problem is not trivial, consider the following snippet of assembly code:

```
1:    (0x000)  LJMP 0x100     ;; Jump forward 256.
2:         ...  ...
3:    (0x0FA)  LJMP 0x100     ;; Jump forward 256.
4:         ...  ...
5:    (0x100)  LJMP -0x100    ;; Jump backward 256.
```

We observe that $100_{16} = 256_{10}$, and lies *just* outside the range expressible in an 8-bit byte (0–255).

As our example shows, given an occurrence l of an `LJMP` instruction, it may be possible to shrink l to an occurrence of an `SJMP`—consuming fewer bytes of code memory—provided we can shrink any `LJMP`s that exist between l and its target location. In particular, if we wish to shrink the `LJMP` occurring at line 1, then we must shrink the `LJMP` occurring at line 3. However, to shrink the `LJMP` occurring at line 3 we must also shrink the `LJMP` occurring at line 5, and *vice versa*.

Further, consider what happens if, instead of appearing at memory address `0x100`, the instruction at line 5 instead appeared *just* beyond the size of code memory, and all other memory addresses were shifted accordingly. Now, in order to be able to successfully fit our program into the MCS-51's limited code memory, we are *obliged* to shrink the `LJMP` occurring at line 5. That is, the shrinking process is not just related to the optimisation of generated machine code but also the completeness of the assembler itself.

How we went about resolving this problem affected the shape of our proof of correctness for the whole assembler in a rather profound way. We first attempted to synthesise a solution bottom up: starting with no solution, we gradually refine a solution using the same functions that implement the jump expansion process. Using this technique, solutions can fail to exist, and the proof of correctness for the assembler quickly descends into a diabolical quagmire.

Abandoning this attempt, we instead split the 'policy'—the decision over how any particular jump should be expanded—from the implementation that actually expands assembly programs into machine code. Assuming the existence of a correct policy, we proved the implementation of the assembler correct. Further, we proved that the assembler fails to assemble an assembly program if and only if a correct policy does not exist. This is achieved by means of dependent types: the assembly function is total over a program, a policy and the proof that the policy is correct for that program.

Policies do not exist in only a limited number of circumstances: namely, if a pseudoinstruction attempts to jump to a label that does not exist, or the program is too large to fit in code memory, even after shrinking jumps according to the best policy. The first circumstance is an example of a serious compiler error, as an ill-formed assembly program was generated, and does not (and should not) count as a mark against the completeness of the assembler. We plan to employ dependent types in CerCo in order to restrict the domain of the compiler to those programs that are 'semantically correct' and should be compiled. In particular, in CerCo we are also interested in the completeness of the compilation process, whereas previous formalisations only focused on correctness.

The rest of this paper is a detailed description of our proof that is, in part, still a work in progress.

1.1 Overview of the paper

In Section 2 we provide a brief overview of the Matita proof assistant for the unfamiliar reader. In Section 3 we discuss the design and implementation of the proof proper. In Section 4 we conclude.

2 Matita

Matita is a proof assistant based on a variant of the Calculus of (Co)inductive Constructions [2]. In particular, it features dependent types that we heavily exploit in the formalisation. The syntax of the statements and definitions in the paper should be self-explanatory, at least to those exposed to dependent type theory. We only remark the use of of '?' or '. . .' for omitting single terms or sequences of terms to be inferred automatically by the system, respectively. Those that are not inferred are left to the user as proof obligations. Pairs are denoted with angular brackets, $\langle -, - \rangle$.

Matita features a liberal system of coercions. It is possible to define a uniform coercion $\lambda x.\langle x, ? \rangle$ from every type T to the dependent product $\Sigma x : T.P\ x$. The coercion opens a proof obligation that asks the user to prove that P holds for x. When a coercion must be applied to a complex term (a λ-abstraction, a local definition, or a case analysis), the system automatically propagates the coercion to the sub-terms For instance, to apply a coercion to force $\lambda x.M : A \to B$ to have type $\forall x : A.\Sigma y : B.P\ x\ y$, the system looks for a coercion from $M : B$ to $\Sigma y : B.P\ x\ y$ in a context augmented with $x : A$. This is significant when the coercion opens a proof obligation, as the user will be presented with multiple, but simpler proof obligations in the correct context. In this way, Matita supports the "Russell" proof methodology developed by Sozeau in [15], with an implementation that is lighter and more tightly integrated with the system than that of Coq.

3 The proof

3.1 The assembler and semantics of machine code

The formalisation in Matita of the semantics of MCS-51 machine code is described in [13]. We merely describe enough here to understand the rest of the proof.

The emulator centres around a `Status` record, describing the microprocessor's state. This record contains fields corresponding to the microprocessor's program counter, registers, and so on. At the machine code level, code memory is implemented as a compact trie of bytes, addressed by the program counter. Machine code programs are loaded into `Status` using the `load_code_memory` function.

We may execute a single step of a machine code program using the `execute_1` function, which returns an updated `Status`:

```
definition execute_1: Status → Status := ...
```

The function `execute` allows one to execute an arbitrary, but fixed (due to Matita's normalisation requirement) number of steps of a program.

Naturally, assembly programs have analogues. The counterpart of the `Status` record is `PseudoStatus`. Instead of code memory being implemented as tries of bytes, code memory is here implemented as lists of pseudoinstructions, and program counters are merely indices into this list. Both `Status` and `PseudoStatus` are specialisations of the same `PreStatus` record, parametric in the representation

of code memory. This allows us to share some code that is common to both records (for instance, 'setter' and 'getter' functions).

Our analogue of `execute_1` is `execute_1_pseudo_instruction`:

```
definition execute_1_pseudo_instruction: (Word → nat × nat) →
                                    PseudoStatus → PseudoStatus := ...
```

Notice, here, that the emulation function for assembly programs takes an additional argument. This is a function that maps program counters (at the assembly level) to pairs of natural numbers representing the number of clock ticks that the pseudoinstruction needs to execute, post expansion. We call this function a *costing*, and note that the costing is induced by the particular strategy we use to expand pseudoinstructions. If we change how we expand conditional jumps to labels, for instance, then the costing needs to change, hence `execute_1_pseudo_instruction`'s parametricity in the costing.

The costing returns *pairs* of natural numbers because, in the case of expanding conditional jumps to labels, the expansion of the 'true branch' and 'false branch' may differ in execution time. This timing information is used inside `execute_1_pseudo_instruction` to update the clock of the `PseudoStatus`. During the proof of correctness of the assembler we relate the clocks of `Status`es and `PseudoStatus`es for the policy induced by the cost model and optimisations.

The assembler, mapping programs consisting of lists of pseudoinstructions to lists of bytes, operates in a mostly straightforward manner. To a degree of approximation, the assembler on an assembly program, consisting of n pseudoinstructions $\mathtt{P_i}$ for $1 \leq i \leq n$, works as in the following diagram (we use $-^*$ to denote a combined map and flatten operation):

$$[\mathtt{P_1}, \ldots \mathtt{P_n}] \xrightarrow{\left(\mathtt{P_i} \xrightarrow{\texttt{expand\_pseudo\_instruction}} [\mathtt{I_i^1}, \ldots \mathtt{I_i^q}] \xrightarrow{\texttt{assembly1}^*} [\mathtt{0110}]\right)^*} [\mathtt{010101}]$$

Here $\mathtt{I_i^j}$ for $1 \leq j \leq q$ are the q machine code instructions obtained by expanding, with `expand_pseudo_instruction`, a single pseudoinstruction P_i. Each machine code instruction $\mathtt{I_j^i}$ is then assembled, using the `assembly1` function, into a list of bytes. This process is iterated for each pseudoinstruction, before the lists are flattened into a single bit list representation of the original assembly program.

By inspecting the above diagram, it would appear that the best way to proceed with a proof that the assembly process does not change the semantics of an assembly program is by proving the same independently for `expand_pseudo_instruction` and for `assembly1`. This is a tempting approach to the proof, but ultimately the wrong approach. In particular, to expand a pseudoinstruction we need to know the address at which the expanded instructions will be located, for instance to know if a short jump is possible. That address is a function of the *machine code* generated for the pseudoinstructions already expanded. Thus, we must assemble each pseudoinstruction into machine code before moving on, and this must be eventually reflected in the proof too. Therefore we will have lemmas proving correctness for `assembly1`, and for the composition of `assembly1` and `expand_pseudo_instruction`, but not for `expand_pseudo_instruction` alone.

3.2 Policies

Policies exist to dictate how conditional and unconditional jumps at the assembly level should be expanded into machine code instructions. Using policies, we are able to completely decouple the decision over how jumps are expanded with the act of expansion, simplifying our proofs. As mentioned, the MCS-51 instruction set includes three different jump instructions: `SJMP`, `AJMP` and `LJMP`; call these 'short', 'medium' and 'long' jumps, respectively:

```
inductive jump_length: Type[0] :=
  |short_jump:jump_length |medium_jump:jump_length |long_jump:jump_length.
```

A `jump_expansion_policy` is a map from pseudo program counters (implemented as `Word`s) to `jump_length`s. Efficient implementations of policies are based on tries. Intuitively, a policy maps positions in a program (indexed using program counters implemented as `Word`s) to a particular variety of jump:

```
definition policy_type := Word → jump_length.
```

Next, we require a series of 'sigma' functions. These functions map assembly program counters to their machine code counterparts, establishing the correspondence between 'positions' in an assembly program and 'positions' in a machine code program. At the heart of this process is `sigma0` which traverses an assembly program building maps from pseudo program counters to program counters. This function fails if and only if an internal call to `assembly_1_pseudoinstruction_safe` fails, which happens if a jump pseudoinstruction is expanded incorrectly:

```
definition sigma0: pseudo_assembly_program → policy_type
  → option (nat × (nat × (BitVectorTrie Word 16))) := ...
```

Here, the returned `BitVectorTrie` is a map between pseudo program counters and program counters, and is constructed by successively expanding pseudoinstructions and incrementing the two program counters the requisite amount to keep them in correct correspondence. The two natural numbers returned are the maximum values that the two program counters attained.

We eventually lift this to functions from pseudo program counters to program counters, implemented as `Word`s:

```
definition sigma_safe:
  pseudo_assembly_program → policy_type → option (Word → Word) := ...
```

Now, it's possible to define what a 'good policy' is for a program `p`. A policy `pol` is deemed good when it prevents `sigma_safe` from failing on `p`. Failure is only possible when the policy dictates that short or medium jumps be expanded to jumps to locations too far away, or when the produced object code does not fit into code memory. A `policy` for a program `p` is a policy that is good for `p`:

```
definition policy_ok := λpol.λp. sigma_safe p ≠ None ...
definition policy :=
  λp. Σjump_expansion: policy_type. policy_ok jump_expansion p
```

Finally, we obtain `sigma`, a mapping from pseudo program counters to program counters that takes in input a good policy and thus never fails. Note how we avoid failure here, and in most of the remaining functions, by restricting the domain using the dependent type `policy`:

```
definition sigma: ∀p. policy p → Word → Word := ...
```

3.3 Correctness of the assembler with respect to fetching

Using our policies, we now work toward proving the total correctness of the assembler. By 'total correctness', we mean that the assembly process never fails when provided with a good policy and that the process does not change the semantics of a certain class of well behaved assembly programs. Naturally, this necessitates keeping some sort of correspondence between addresses at the assembly level and addresses at the machine code level. For this, we use the `sigma` machinery defined at the end of Subsection 3.2.

We expand pseudoinstructions using the function `expand_pseudo_instruction`. This takes an assembly program (consisting of a list of pseudoinstructions), a good policy for the program and a pointer to the pseudo code memory. It returns a list of instructions, corresponding to the expanded pseudoinstruction referenced by the pointer. The policy is used to decide how to expand `Call`s, `Jmp`s and conditional jumps. The function is given a dependent type that incorporates its specification. Its pre- and post-conditions are omitted in the paper due to lack of space. We show them as an example in the next function, `build_maps`.

```
definition expand_pseudo_instruction:
  ∀program. ∀pol: policy program.
  ∀ppc:Word. ... Σres. list instruction. ... := ...
```

The following function, `build_maps`, is used to construct a pair of mappings from program counters to labels and cost labels, respectively. Cost labels are a technical device used in the CerCo prototype C compiler for proving that the compiler is cost preserving. For our purposes in this paper, they can be safely ignored, though the interested reader may consult [1] for an overview.

The label map, on the other hand, records the position of labels that appear in an assembly program, so that the pseudoinstruction expansion process can replace them with real memory addresses:

```
definition build_maps:
 ∀p. ∀pol: policy p.
 Σres : ((BitVectorTrie Word 16) × (BitVectorTrie Word 16)).
   let ⟨labels, costs⟩ := res in
     ∀id. occurs_exactly_once id (π2 p) →
    let addr := address_of_word_labels_code_mem (π2 p) id in
      lookup ... id labels (zero ...) = sigma pseudo_program pol addr := ...
```

The type of `build_maps` owes to our use of Matita's Russell facility to provide a strong specification for the function in the type (c.f. the use of Σ-types and

coercions, through which Russell is simulated in Matita). We express that for all labels that appear exactly once in any assembly program, the newly created map used in the implementation, and the stronger `sigma` function used in the specification, agree.

Using `build_maps`, we can express the following lemma, expressing the correctness of the assembly function:

```
lemma assembly_ok: ∀p,pol,assembled.
  let ⟨labels, costs⟩ := build_maps p pol in
  ⟨assembled,costs⟩ = assembly p pol →
  let cmem := load_code_memory assembled in
  let preamble := π1 p in
  let dlbls := construct_datalabels preamble in
  let addr := address_of_word_labels_code_mem (π2 p) in
  let lk_lbls := λx. sigma p pol (addr x) in
  let lk_dlbls := λx. lookup … x datalabels (zero ?) in
  ∀ppc, pi, newppc.
  ∀prf: ⟨pi, newppc⟩ = fetch_pseudo_instruction (π2 p) ppc.
  ∀len, assm.
  let spol := sigma program pol ppc in
  let spol_len := spol + len in
  let echeck := encoding_check cmem spol spol_len assm in
  let a1pi := assembly_1_pseudoinstruction in
  ⟨len, assm⟩ = a1pi p pol ppc lk_lbls lk_dlbls pi (refl …) (refl …) ? →
    echeck ∧ sigma p pol newppc = spol_len.
```

Suppose also we assemble our program `p` in accordance with a policy `pol` to obtain `assembled`. Here, we perform a 'sanity check' to ensure that the two cost label maps generated are identical, before loading the assembled program into code memory `cmem`. Then, for every pseudoinstruction `pi`, pseudo program counter `ppc` and new pseudo program counter `newppc`, such that we obtain `pi` and `newppc` from fetching a pseudoinstruction at `ppc`, we check that assembling this pseudoinstruction produces the correct number of machine code instructions, and that the new pseudo program counter `ppc` has the value expected of it.

Theorem `fetch_assembly` establishes that the `fetch` and `assembly1` functions interact correctly. The `fetch` function, as its name implies, fetches the instruction indexed by the program counter in the code memory, while `assembly1` maps a single instruction to its byte encoding:

```
theorem fetch_assembly: ∀pc, i, cmem, assembled. assembled=assembly1 i →
  let len := length … assembled in
  encoding_check cmem pc (pc + len) assembled →
    let fetched := fetch code_memory (bitvector_of_nat … pc) in
    let ⟨instr_pc, ticks⟩ := fetched in
    let ⟨instr, pc'⟩ := instr_pc in
      (eq_instruction instr i ∧ eqb ticks (ticks_of_instruction instr) ∧
       eq_bv … pc' (pc + len)) = true.
```

In particular, we read `fetch_assembly` as follows. Given an instruction, `i`, we first assemble the instruction to obtain `assembled`, checking that the assembled

instruction was stored in code memory correctly. Fetching from code memory, we obtain `fetched`, a tuple consisting of the instruction, new program counter, and the number of ticks this instruction will take to execute. Deconstructing these tuples, we finally check that the fetched instruction is the same instruction that we began with, and the number of ticks this instruction will take to execute is the same as the result returned by a lookup function, `ticks_of_instruction`, devoted to tracking this information. Or, in plainer words, assembling and then immediately fetching again gets you back to where you started.

Lemma `fetch_assembly_pseudo` (slightly simplified, here) is obtained by composition of `expand_pseudo_instruction` and `assembly_1_pseudoinstruction`:

```
lemma fetch_assembly_pseudo:
 ∀program.∀pol:policy program.∀ppc.∀code_memory.
  let pi := π1 (fetch_pseudo_instruction (π2 program) ppc) in
  let instructions := expand_pseudo_instruction program pol ppc ...in
  let ⟨len,assembled⟩ := assembly_1_pseudoinstruction program pol ppc ...in
  encoding_check code_memory pc (pc + len) assembled →
  fetch_many code_memory (pc + len) pc instructions.
```

Here, `len` is the number of machine code instructions the pseudoinstruction at hand has been expanded into, and `encoding_check` is a recursive function that checks that assembled machine code is correctly stored in code memory. We assemble a single pseudoinstruction with `assembly_1_pseudoinstruction`, which internally calls `jump_expansion` and `expand_pseudo_instruction`. The function `fetch_many` fetches multiple machine code instructions from code memory and performs some routine checks.

Intuitively, Lemma `fetch_assembly_pseudo` can be read as follows. Suppose we expand the pseudoinstruction at `ppc` with the policy decision `pol`, obtaining the list of machine code instructions `instructions`. Suppose we also assemble the pseudoinstruction at `ppc` to obtain `assembled`, a list of bytes. Then, we check with `fetch_many` that the number of machine instructions that were fetched matches the number of instruction that `expand_pseudo_instruction` expanded.

The final lemma in this series is `fetch_assembly_pseudo2` that combines the Lemma `fetch_assembly_pseudo` with the correctness of the functions that load object code into the processor's memory.

```
lemma fetch_assembly_pseudo2:
 ∀program,pol,ppc.
  let ⟨labels,costs⟩ := build_maps program pol in
  let assembled := π1 (assembly program pol) in
  let code_memory := load_code_memory assembled in
  let data_labels := construct_datalabels (π1 program) in
  let lookup_labels :=
    λx. sigma ... pol (address_of_word_labels_code_mem (π2 program) x) in
  let lookup_datalabels := λx. lookup ? ? x data_labels (zero ?) in
  let ⟨pi,newppc⟩ := fetch_pseudo_instruction (π2 program) ppc in
  let instrs :=expand_pseudo_instruction program pol ppc ...in
   fetch_many code_memory (sigma ... pol newppc) (sigma ... pol ppc) instrs.
```

We read `fetch_assembly_pseudo2` as follows. Suppose we are able to successfully assemble an assembly program using `assembly` and produce a code memory, `code_memory`. Then, fetching a pseudoinstruction from the pseudo code memory at address `ppc` corresponds to fetching a sequence of instructions from the real code memory at address `sigma program pol ppc`. The fetched sequence corresponds to the expansion, according to `pol`, of the pseudoinstruction.

At first, the lemmas appears to immediately imply the correctness of the assembler. However, this property is *not* strong enough to establish that the semantics of an assembly program has been preserved by the assembly process since it does not establish the correspondence between the semantics of a pseudoinstruction and that of its expansion. In particular, the two semantics differ on instructions that *could* directly manipulate program addresses.

3.4 Total correctness for 'well behaved' assembly programs

In any 'reasonable' assembly language addresses in code memory are just data that can be manipulated in multiple ways by the program. An assembly program can forge, compare and move addresses around, shift existing addresses or apply logical and arithmetical operations to them. Further, every optimising assembler is allowed to modify code memory. Hence only the semantics of a few of the aforementioned operations are preserved by an optimising assembler/compiler. Moreover, this characterisation of well behaved programs is clearly undecidable.

To obtain a reasonable statement of correctness for our assembler, we need to trace memory locations (and, potentially, registers) that contain memory addresses. This is necessary for two purposes.

First we must detect (at run time) programs that manipulate addresses in well behaved ways, according to some approximation of well-behavedness. Second, we must compute statuses that correspond to pseudo-statuses. The contents of the program counter must be translated, as well as the contents of all traced locations, by applying the `sigma` map. Remaining memory cells are copied *verbatim*.

For instance, after a function call, the two bytes that form the return pseudo address are pushed on top of the stack, i.e. in internal RAM. This pseudo internal RAM corresponds to an internal RAM where the stack holds the real addresses after optimisation, and all the other values remain untouched.

We use an `internal_pseudo_address_map` to trace addresses of code memory addresses in internal RAM. The current code is parametric on the implementation of the map itself.

```
axiom internal_pseudo_address_map: Type[0].
```

The `low_internal_ram_of_pseudo_low_internal_ram` function converts the lower internal RAM of a `PseudoStatus` into the lower internal RAM of a `Status`. A similar function exists for higher internal RAM. Note that both RAM segments are indexed using addresses 7-bits long. The function is currently axiomatised, and an associated set of axioms prescribe the behaviour of the function:

```
axiom low_internal_ram_of_pseudo_low_internal_ram:
```

```
internal_pseudo_address_map→BitVectorTrie Byte 7→BitVectorTrie Byte 7.
```

Next, we are able to translate `PseudoStatus` records into `Status` records using `status_of_pseudo_status`. Translating a `PseudoStatus`'s code memory requires we expand pseudoinstructions and then assemble to obtain a trie of bytes. This never fails, providing that our policy is correct:

```
definition status_of_pseudo_status: internal_pseudo_address_map →
  ∀ps:PseudoStatus. policy (code_memory ... ps) → Status
```

The `next_internal_pseudo_address_map` function is responsible for run time monitoring of the behaviour of assembly programs, in order to detect well behaved ones. It returns a map that traces memory addresses in internal RAM after execution of the next pseudoinstruction, failing when the instruction tampers with memory addresses in unanticipated (but potentially correct) ways. It thus decides the membership of a strict subset of the set of well behaved programs.

```
definition next_internal_pseudo_address_map: internal_pseudo_address_map
  → PseudoStatus → option internal_pseudo_address_map
```

The function `ticks_of` computes how long—in clock cycles—a pseudoinstruction will take to execute when expanded in accordance with a given policy. The function returns a pair of natural numbers, needed for recording the execution times of each branch of a conditional jump.

```
definition ticks_of:
  ∀p:pseudo_assembly_program. policy p → Word → nat × nat := ...
```

Finally, we are able to state and prove our main theorem. This relates the execution of a single assembly instruction and the execution of (possibly) many machine code instructions, as long . That is, the assembly process preserves the semantics of an assembly program, as it is translated into machine code, as long as we are able to track memory addresses properly:

```
theorem main_thm:
 ∀M,M':internal_pseudo_address_map.∀ps.∀pol: policy ps.
  next_internal_pseudo_address_map M ps = Some ... M' →
   ∃n.
     execute n (status_of_pseudo_status M ps pol)
    = status_of_pseudo_status M'
       (execute_1_pseudo_instruction (ticks_of (code_memory ... ps) pol) ps)
       [pol].
```

The statement is standard for forward simulation, but restricted to `PseudoStatuses ps` whose next instruction to be executed is well-behaved with respect to the `internal_pseudo_address_map M`. Theorem `main_thm` establishes the total correctness of the assembly process and can simply be lifted to the forward simulation of an arbitrary number of well behaved steps on the assembly program.

4 Conclusions

We are proving the total correctness of an assembler for MCS-51 assembly language. In particular, our assembly language featured labels, arbitrary conditional and unconditional jumps to labels, global data and instructions for moving this data into the MCS-51's single 16-bit register. Expanding these pseudoinstructions into machine code instructions is not trivial, and the proof that the assembly process is 'correct', in that the semantics of a subset of assembly programs are not changed is complex. Further, we have observed the 'shocking' fact that any optimising assembler cannot preserve the semantics of all assembly programs.

The formalisation is a key component of the CerCo project, which aims to produce a verified concrete complexity preserving compiler for a large subset of the C programming language. The verified assembler, complete with the underlying formalisation of the semantics of MCS-51 machine code (described fully in [13]), will form the bedrock layer upon which the rest of the CerCo project will build its verified compiler platform. However, further work is needed. In particular, as it stands, the code produced by the prototype CerCo C compiler does not fall into the 'semantics preserving' subset of assembly programs for our assembler. This is because the MCS-51 features a small stack space, and a larger stack is customarily manually emulated in external RAM. As a result, the majority of programs feature slices of memory addresses and program counters being moved in-and-out of external RAM via the registers, simulating the stack mechanism. At the moment, this movement is not tracked by `internal_pseudo_address_map`, which only tracks the movement of memory addresses in low internal RAM. We leave extending this tracking of memory addresses throughout the whole of the MCS-51's address spaces as future work.

It is interesting to compare our work to an 'industrial grade' assembler for the MCS-51: SDCC [14]. SDCC is the only open source C compiler that targets the MCS-51 instruction set. It appears that all pseudojumps in SDCC assembly are expanded to `LJMP` instructions, the worst possible jump expansion policy from an efficiency point of view. Note that this policy is the only possible policy *in theory* that can preserve the semantics of an assembly program during the assembly process. However, this comes at the expense of assembler completeness: the generated program may be too large to fit into code memory. In this respect, there is a trade-off between the completeness of the assembler and the efficiency of the assembled program. The definition and proof of a complete, optimal (in the sense that object code size is minimised) and correct jump expansion policy is ongoing work.

Aside from their application in verified compiler projects such as CerCo and CompCert, verified assemblers such as ours could also be applied to the verification of operating system kernels. Of particular note is the verified seL4 kernel [6,7]. This verification explicitly assumes the existence of, amongst other things, a trustworthy assembler and compiler.

Note that both CompCert and the seL4 formalisation assume the existence of 'trustworthy' assemblers. Our observation that an optimising assembler cannot preserve the semantics of every assembly program may have important conse-

quences for these projects. If CompCert chooses to assume the existence of an optimising assembler, then care should be made to ensure that any assembly program produced by the CompCert compiler falls into the subset of programs that have a hope of having their semantics preserved by an optimising assembler.

Our formalisation exploits dependent types in different ways and for multiple purposes. The first purpose is to reduce potential errors in the formalisation of the microprocessor. In particular, dependent types are used to constraint the size of bit-vectors and tries that represent memory quantities and memory areas respectively. They are also used as explained in [13]. to simulate polymorphic variants in Matita. Polymorphic variants nicely capture the absolutely unorthogonal instruction set of the MCS-51 where every opcode must accept its own subset of the 11 addressing mode of the processor.

The second purpose is to single out the sources of incompleteness. By abstracting our functions over the dependent type of correct policies, we were able to manifest the fact that the compiler never refuses to compile a program where a correct policy exists. This also allowed to simplify the initial proof by dropping lemmas establishing that one function fails if and only if some other one does so.

Finally, dependent types, together with Matita's liberal system of coercions, allow to simulate almost entirely in user space the proof methodology "Russell" of Sozeau [15]. However, not every proof has been done this way: we only used this style to prove that a function satisfies a specification that only involves that function in a significant way. For example, it would be unnatural to see the proof that fetch and assembly commute as the specification of one of the two functions.

4.1 Related work

We are not the first to consider the total correctness of an assembler for a non-trivial assembly language. Perhaps the most impressive piece of work in this domain is the Piton stack [11,12]. This was a stack of verified components, written and verified in ACL2, ranging from a proprietary FM9001 microprocessor verified at the gate level, to assemblers and compilers for two high-level languages—a dialect of Lisp and μGypsy [12].

Klein and Nipkow consider a Java-like programming language, Jinja [8,9]. They provide a compiler, virtual machine and operational semantics for the programming language and virtual machine, and prove that their compiler is semantics and type preserving.

We believe some other verified assemblers exist in the literature. However, what sets our work apart from that above is our attempt to optimise the machine code generated by our assembler. This complicates any formalisation effort, as the best possible selection of machine instructions must be made, especially important on a device such as the MCS-51 with a miniscule code memory. Further, care must be taken to ensure that the time properties of an assembly program are not modified by the assembly process lest we affect the semantics of any program employing the MCS-51's I/O facilities. This is only possible by inducing a cost model on the source code from the optimisation strategy and input program. This will be a *leit motif* of CerCo.

Finally, mention of CerCo will invariably invite comparisons with CompCert [5,10], another verified compiler project related to CerCo. As previously mentioned, CompCert considers only extensional correctness of the compiler, and not intensional correctness, which CerCo focusses on. However, CerCo also extends CompCert in other ways. Namely, the CompCert verified compilation chain terminates at the assembly level, and takes for granted the existence of a trustworthy assembler. CerCo chooses to go further, by considering a verified compilation chain all the way down to the machine code level. The work presented in this publication is one part of CerCo's extension over CompCert.

4.2 Resources

All files relating to our formalisation effort can be found online at `http://cerco.cs.unibo.it`. The code of the compiler has been completed, and the proof of correctness described here is still in progress. In particular, we have assumed several properties of "library functions" related in particular to modular arithmetics and datastructures manipulation. Moreover, we only completed the interesting cases of some of the main theorems that proceed by cases on all the possible opcodes. We thus believe that the proof strategy is sound and that we will be able to close soon all axioms, up to possible minor bugs that should have local fixes that do not affect the global proof strategy.

The development, including the definition of the executable semantics of the MCS-51, is spread across 17 files, totalling around 11,500 lines of Matita source. The bulk of the proof described herein is contained in a single file, `AssemblyProof.ma`, consisting at the moment of approximately 2500 lines of Matita source. Another 1000 lines of proofs are spread all over the development because of dependent types and the Russell proof style, that do not allow to separate the code from the proofs. The low ratio between the number of lines of code and the number of lines of proof is unusual. It is justified by the fact that the pseudo-assembly and the assembly language share most constructs and that large parts of the semantics is also shared. Thus many lines of code are required to describe the complex semantics of the processor, but, for the shared cases, the proof of preservation of the semantics is essentially trivial.

References

1. Amadio, R.M., Ayache, N., Régis-Gianas, Y., Saillard, R.: Cerifying cost annotations in compilers. Tech. rep., Université Paris Diderot (Paris 7), Laboratoire PPS (2010)
2. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Automated Reasoning 39, 109–139 (2007)
3. Bate, I., Khan, U.: WCET analysis of modern processors using multi-criteria optimisation. Empirical Software Engineering 16, 5–28 (2011)
4. The CerCo FET-Open project. `http://cerco.cs.unibo.it/` (2011)
5. The CompCert project. `http://compcert.inria.fr/` (2011)
6. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Thomas Sewell, H.T., Winwood, S.: seL4: Formal verification of an operating system kernel. In: SOSP (2009)

7. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Thomas Sewell, H.T., Winwood, S.: seL4: Formal verification of an operating system kernel. CACM 53, 107–115 (2010)
8. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. TOPLAS 28(4), 619–695 (2006)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. Tech. Rep. 0400001T.1, National ICT Australia (2010)
10. Leroy, X.: Formal verification of a realistic compiler. CACM 52(7), 107–115 (2009)
11. Moore, J.S.: Piton: A mechanically verified assembly language, Automated Reasoning Series, vol. 3. Springer (1996)
12. Moore, J.S.: A grand challenge proposal for formal methods (2005)
13. Mulligan, D.P., Sacerdoti Coen, C.: An executable formal semantics of the MCS-51 microprocessor in Matita. In: FMCAD (2011), submitted
14. Small device C compiler 3.0.0. http://sdcc.sourceforge.net/ (2011)
15. Sozeau, M.: Subset coercions in Coq. In: TYPES. pp. 237–252 (2006)