# Towards a Framework for Developing Verified Assemblers for the ELF Format

Jinhua Wu[1], Yuting Wang[1](✉), Meng Sun[1], Xiangzhe Xu[2], and Yichen Song[1]

[1] Shanghai Jiao Tong University, Shanghai, China
yuting.wang@sjtu.edu.cn
[2] Purdue University, West Lafayette, USA
xu1415@purdue.edu

**Abstract.** Most of the existing work on verified compilation leaves unverified the translation of assembly programs into binary code in object file formats (e.g., the Executable and Linkable Format or ELF). The challenges of developing verified assemblers come from the intrinsic complexities in low-level assembling processes caused by the need to support different computer architectures and their details, such as encoding a large number of instructions and verifying its correctness. We present a framework that overcomes the above challenges. It works as a template which may be instantiated to generate verified assemblers for different architectures targeting ELF object files. For this, it is parameterized over the implementation and verification of architecture-dependent assembling processes through well-defined interfaces. By plugging the architecture-dependent parts into the template, we get complete verified assemblers. To manage the complexity in developing and verifying encoding of instructions, we integrate into our framework the CSLED framework for automatically generating verified instruction encoders and decoders from declarative instruction specifications. To show the effectiveness of our framework, we have applied it to generate verified assemblers for the complete X86 and RISC-V assembly languages in CompCert.

## 1 Introduction

Although the formal development of compilers and their correctness proofs have been extensively studied (e.g. the state-of-the-art verified C compiler CompCert [7,8]), few of the existing work has completed the last mile, i.e., to verify the translation of assembly code into machine code. An obvious obstacle of developing verified assemblers is the potentially large amount of work to support different commercial architectures. Even for a single architecture, the details need to be taken care of during assembly are overwhelming. A typical example is the encoding of assembly instructions into machine instructions which may be hundreds or even thousands in number in any *instruction set architecture* (ISA).

To manage the high complexity in building assemblers that target different object files formats (e.g., PE/COFF, Mach-O and ELF) and architectures


C.-K. Hur (Ed.): APLAS 2023, LNCS 14405, pp. 205–224, 2023.
https://doi.org/10.1007/978-981-99-8311-7_10

(e.g., X86, RISC-V and ARM), the standard practice in industry is to separate the implementation of platform-independent parts of assemblers from the platform-dependent parts. The GNU assembler [18] follows this approach by employing Binary File Descriptor (BFD) to implement this separation. The same idea should also be applicable to verified assemblers. However, the existing work on assembler verification does not provide this flexibility as they are designed to work for ad-hoc machine code formats or for fixed architectures (see Sect. 6).

In this paper, we present our initial attempt to develop a framework for building verified assemblers that target the ELF format by following the above idea. In our framework, the architecture-independent parts of assemblers are developed separately from the architecture-dependent parts. The former is captured by a template of implementation and proofs which formalizes the assembly processes that transform the architecture-independent parts of assembly programs into constituents of ELF objects (e.g., generation of symbol tables). Furthermore, this template is parameterized over the architecture-dependent transformations (e.g., instruction encoding) through well-defined interfaces. To generate a verified assembler for a specific architecture, users instantiate the template with the implementation and proofs of the architecture-dependent components for this architecture through these interfaces. An immediate benefit of this approach is the ability to generate assemblers targeting different platforms by only switching the architecture-dependent instances, which significantly reduces the complexity in developing verified assemblers.

An essential architecture-dependent assembly pass is the encoding of assembly instructions into binary machine code. It is difficult to implement and even more difficult to prove correct because there are at least hundreds or sometimes even thousands of instructions in a common ISA. To tackle this difficulty, we adopt the CSLED framework [22]. In CSLED, one can write down an instruction format as a declarative specification, from which a pair of verified instruction encoder and decoder is automatically generated. However, the generated encoders and decoders work with a form of abstract assembly instructions different from our source assembly language. We address this problem by developing verified translators to connect the verified encoders with our assembly instructions.

Our framework is implemented in the Coq proof assistant [17] and utilizes CompCert's infrastructure [7]. To demonstrate its effectiveness, we apply it to generate verified assemblers for the complete X86 and RISC-V assembly languages used in CompCert. There are different challenges to apply our framework to X86 and RISC-V. For X86, we need to deal with the complex instruction format. For RISC-V, we improve the CSLED framework to overcome the limitation in CSLED for supporting RISC instructions. To examine their usefulness, we have connected them with the back-end of the newest version of Stack-Aware CompCert [19,21] to form a full compilation chain from C to ELF objects. We choose to connect with Stack-Aware CompCert instead of the regular CompCert because its target assembly languages are closer to realistic assembly languages (e.g., no pseudo instructions for stack manipulation). Note that this

connection is not fully verified yet due to limitations in (Stack-Aware) CompCert (see Sect. 5.1).

We summarize our contributions as follows:

- Our key contribution is an approach to developing customizable verified assemblers targeting different ISAs by separating the architecture-independent and -dependent components of verified assemblers, such that the former are abstracted over the latter through well-defined interfaces. To generate concrete verified assemblers, users only need to provide instances of architecture-dependent components which meet the abstract interfaces. The design of such interfaces is a key challenge of this work. To reduce the effort for instantiating instruction encoders, we integrate the automation framework CSLED into our framework. Users only need to write down declarative specifications, from which verified encoders are automatically generated, and add glue code to integrate these encoders into verified assemblers.
- We demonstrate the effectiveness and flexibility of our approach by applying our framework to develop verified assemblers for the complete X86 and RISC-V assembly languages in CompCert. We have successfully replaced the unverified GNU assembler used by CompCert with our verified assemblers, therefore significantly reduce its TCB. These applications show that the complexity of implementing verified assemblers for different ISAs is confined in architecture-dependent components, and it takes a reasonable amount of effort to support representative CISC and RISC architectures.

The entire framework and their applications can be found at https://doi.org/10.5281/zenodo.8363543. In the rest of the paper, we first introduce necessary background in Sect. 2. We then present the design of our framework in Sect. 3. In Sect. 4, we discuss the application of our framework to X86 and RISC-V. We connect the instantiated assemblers to Stack-Aware CompCert and discuss evaluation in Sect. 5. Finally, we discuss the related work and conclude in Sect. 6.

## 2 Background

### 2.1 A Running Example

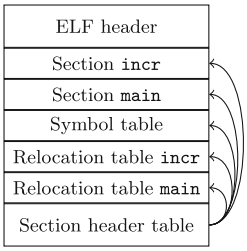
To provide a better understanding of the background knowledge, we introduce a running example which is a simple C program that gets compiled to X86-64 assembly code and finally translated into an ELF object file, as shown in Fig. 1. In the C program, `main` initializes the global variable `counter` and calls `incr` to increase it by one. The corresponding assembly code is in the AT&T X86 syntax, in which `incr` loads `counter` into the register `eax`, adds one to `eax` by using `leal` instruction, and then stores the modified value back to the `counter`. `counter` is labeled as a *common* symbol which is not initialized and not allocated in the object file. Note that we have omitted instructions not relevant to our discussion (e.g., for stack allocation). In the later sections, the running example will be used to explain the important concepts and components of our framework, such as generation of symbols and sections, generation of relocation information for linking, and encoding of instructions.

```
1  int counter;
2
3  void incr(){
4    counter++;
5  }
6
7  int main(){
8    counter = 0;
9    incr();
10 }
```

(a) C code

```
counter: # common symbol
incr:
  ...
  movl counter,%eax
  leal 1(%eax),%eax
  movl %eax,counter
  ...
main:
  ...
  call incr
  ...
```

(b) Assembly code




(c) ELF object file

**Fig. 1.** A Running Example

### 2.2 Compiler Verification Based on Simulation

Correctness of compilation is often described as preservation of program semantics. A common approach to semantics preservation is to model semantics in a small-step style as *labeled transition systems* (LTS) and to establish *simulation relations* between the source and target semantics. Our framework makes use of this approach, in particular, its realization in CompCert [7] which consists of a sequence of passes that successively translate a large subset of C into assembly languages. We discuss the essential concepts supporting this approach below.

CompCert provides a uniform abstraction of programs for all of its languages. In any language $\mathcal{L}$ of CompCert, a program $P$ of type $P_{\mathcal{C}}$ consists of a mapping from identifiers to global definitions which are parameterized by the types of functions and information of variables (denoted by $F$ and $V$, respectively):

$$G := \lambda(F\ V : \mathit{Type}).\langle \mathit{fun} : F,\ \mathit{var} : \mathit{Gv}\ V\rangle$$
$$P_{\mathcal{C}} := \lambda(F\ V : \mathit{Type}).\{\mathit{defs} : \mathit{List}\ (\mathit{id} * (G\ F\ V))\}.$$

Here, we use $\{\cdot\}$ to represent records and $\langle\cdot\rangle$ to represent variants. *id* is the type of identifiers. $G$ is the type of global definitions, which may be either functions of type $F$ or variables of type $\mathit{Gv}\ V$ where $\mathit{Gv}$ provides information about variables such as their initial values. For instance, the formalized C program in the running example contains three global definitions: two functions and one variable.

Assembly programs are based on this uniform representation, albeit parameterized by the type of instructions to support different architectures:

$$\mathit{Fn} := \lambda(I : \mathit{Type}).\{\mathit{signature} : \mathit{Sig},\ \mathit{code} : \mathit{List}\ I\}$$
$$\mathit{Fd} := \lambda(I : \mathit{Type}).\langle \mathit{internal} : \mathit{Fn}\ I,\ \mathit{external} : \mathit{Ef}\rangle$$
$$P_{\mathcal{A}} := \lambda(I : \mathit{Type}).P_{\mathcal{C}}\ (\mathit{Fd}\ I)\ \mathit{Unit}.$$

Here, $I$ is the type parameter of instructions. A function of type $\mathit{Fd}$ is either internal or external, where an internal function (of type $\mathit{Fn}$) has a signature and consists of a list of assembly instructions. The type *Unit* denotes that there is no type information for global variables in assembly programs. For instance,

the formalized assembly program in the running example again contains two functions and one variable where the functions are parameterized by an inductive definition of X86 instructions.

Memory models are essential components of program semantics. CompCert adopts a uniform memory model for all of its languages [9,10]. A memory state $m$ consists of a finite set of memory blocks with distinct identifiers and with linear memory addresses such that $(b, \delta)$ denotes a pointer to block $b$ at address $\delta$. Such abstraction enables straightforward pointer arithmetic and memory isolation which are essential for low-level programming. With the uniform memory model, the semantics of a program $P$ of type $(P_{\mathcal{C}}\ F\ V)$ is defined as an LTS derived from the following relations over program states which are pairs of memory states of type $M$ and language-specific states of type $St$ (e.g., register states in assembly programs). Moreover, $Tc$ is the type of event traces, and $Prop$ is the type of propositions.

$$\begin{aligned} &init : \lambda(F\ V : Type).(P_{\mathcal{C}}\ F\ V) \rightarrow (M \times St) \rightarrow Prop \\ &step : (M \times St) \rightarrow Tc \rightarrow (M \times St) \rightarrow Prop. \end{aligned}$$

Here, *init* establishes the initial program state as a result of loading $P$; *step* describes the effect of one-step execution which emits a list of events. The memory initialized by *init* contains a unique block for each global definition in $P$. In the remaining discussions, we denote the semantics of $P$ in language $\mathcal{L}$ as $[\![P]\!]_{\mathcal{L}}$, or simply $[\![P]\!]$ if $\mathcal{L}$ can be inferred from the context.

For a given compiler pass $\mathbb{C}$ described as a partial function, if $\mathbb{C}(P) = \lfloor P' \rfloor$ (where $\lfloor \cdot \rfloor$ is the *some* constructor of the *option* type), CompCert establishes a *forward simulation* between $[\![P]\!]$ and $[\![P']\!]$ denoted by $[\![P]\!] \preccurlyeq [\![P']\!]$. A particular instance we will use in this paper is the *lock-step forward simulation*, for which an *invariant* (or simulation relation) $\sim$ between source and target program states is defined and satisfies the following conditions: *1)* $(m, s) \sim (m', s')$ holds for the initial states $(m, s)$ and $(m', s')$, and *2)* $\sim$ is preserved during the execution. We write $[\![P]\!] \preccurlyeq_{\sim} [\![P']\!]$ when $\sim$ is explicitly given. Note that $\sim$ must capture the relation between source and target memory states which is represented by *memory injections* [9,10]. A memory injection $j$ is a partial function which maps source memory blocks into target blocks. The values (e.g., pointers) stored in the source and target memory must be related according to the injection. A special case is when $\sim$ is equality, meaning that the injection is an identity function. We shall write $[\![P]\!] \preccurlyeq_{=} [\![P']\!]$ to denote simulations with the equality invariant.

With the above definitions, the correctness of $\mathbb{C}$ is formulated as follows:

$$\forall P\ P', \mathbb{C}(P) = \lfloor P' \rfloor \Longrightarrow [\![P]\!] \preccurlyeq [\![P']\!].$$

By vertically composing simulations established for every compiler pass, the semantics preservation of CompCert is proved.

### 2.3 Relocatable ELF Object Files

The verified assemblers we intend to develop target relocatable ELF object files, which represent open binary modules that may be linked into executable ELF

programs. As shown in Fig. 1c, a relocatable ELF object consists of an ELF header which contains meta-information, a list of sections containing program data (including symbol and relocation tables), and section header tables that store the attributes of these sections (e.g., locations of sections in the object).

Sections are the key constituents of ELF objects. In this work, we are only concerned with four kinds of sections: code sections, data sections, symbol tables, and relocation tables. Code and data sections store the binary form of instructions and data. In our running example, the assembly program is complied to an object with two code sections for `incr` and `main`, respectively. It has no data section for `counter` as it is not needed for global variables with no initial value. Symbol tables are used to record references to global definitions. A symbol table consists of a list of *symbol entries*. Each entry contains the information extracted from a global definition in the program, including the type of the definition (e.g., function or data), the type of its binding (e.g., local or global), the section index which points to the section where the definition resides in (special indices are used for common or external symbols), its value (e.g., the offset into its section) and size. In Fig. 1c, there is a single symbol table containing three symbol entries for the global definitions, where `counter` is labeled as a common symbol.

A code or data section may refer to symbols whose addresses cannot be resolved at compile time (e.g., any reference to global definitions in a section whose memory location may be adjusted by the linker). In this case, there is a relocation table associated with this section which consists of *relocation entries*. Each relocation entry points to a location in the section that stores an unresolved symbol. During the linking, the linker would determine the concrete addresses of these symbols and overwrite this location with them. More specifically, a *relocation entry* contains the offset of the unresolved symbol in its section, the relocation type (e.g., relative or absolute addressing), the identifier of the unresolved symbol, and a constant addend for adjusting the symbol address. In our example, the addresses (or relative addresses) of `incr` and `counter` are unknown before linking. Therefore, there are two relocation tables for the sections for `incr` and `main`, respectively. The table for `incr` contains two relocation entries pointing to `counter` in `movl counter,%eax` and `movl %eax,counter`, and `main` contains one entry for `call incr`. The linker will determine the addresses of `incr` and `counter` and overwrite the locations pointed by the relocation entries.

### 2.4 Machine Instruction Formats

A main job of assemblers is to encode assembly instructions into their binary form. For this, we need to understand the binary format of instructions. In this paper, we are concerned with two representative CISC and RISC instruction formats, i.e., X86 and RISC-V instruction formats.

**X86.** Figure 2 shows the format of X86 instructions. An instruction consists of a sequence of binary tokens. The REX prefixes, when present, indicate the instructions are in 64-bit mode or in 32-bit mode and referring to extended

| REX Prefix | Opcode | ModRM | SIB | Disp | Imms |
|---|---|---|---|---|---|

**Fig. 2.** The Format of X86 Instructions

registers (`r8` to `r15`). An opcode is 1 to 3 bytes in length and determines the types of instructions. The ModRM byte indicates which addressing modes are used for the operands of this instruction. These addressing modes, which follow the ModRM byte, include SIB (Scale, Index, and Base) byte and a displacement of the address of the referred symbol. For an instruction operating on immediate values, a token of immediate data (Imms) must occur at the end of it.

We use the instruction `movl counter,%eax` in the running example and its variants `movq counter,%rax` and `movl counter,%r8` to demonstrate instruction encoding. For `movl counter,%eax`, its encoding in hexadecimal is `{Opcode:8B, ModRM:05, Disp:00 00 00 00}`. Here, `8B` is the opcode for move instructions that move memory contents to a register. `05` contains the encoding of `eax` and part of the addressing mode. `Disp` is the location that stores the address of `counter` which is currently zero and to be resolved by linking. The encoding of `movq counter,%rax` has an REX prefix `48` which indicates it is a 64-bit instruction. `movl counter,%r8` is 32-bit albeit refers to an extended register `r8`. It has an REX prefix `44` which contains one bit in the encoded `r8` because `r8` requires four bits to encode but there is only space for three bits in `ModRM`.

**RISC-V.** RISC-V instructions have a uniform size of 32-bit. Therefore, each instruction consists of a single token. RISC-V uses different formats for different types of instructions. Their encoding is straightforward because, given any instruction of a specific type, the positions of its operands are fixed by its format.

### 2.5 The CSLED Framework

To alleviate the difficulty of the instruction encoding, we employ the CSLED framework. CSLED [22] is a meta-programming framework for automatic generation of verified instruction encoders and decoders from declarative specifications of instruction formats. Given an instruction set, the user first writes down its specifications $\mathcal{S}$ in the CSLED instruction specification language, which capture the encoding format of the instruction set (e.g., the X86 format described in Sect. 2.4). Given $\mathcal{S}$, the framework generates an abstract syntax of instructions $\mathbb{A}$, an encoder $\mathbb{E} : \mathbb{A} \rightarrow \lfloor \mathit{List\ Byte} \rfloor$ and a decoder $\mathbb{D} : \mathit{List\ Byte} \rightarrow \lfloor \mathbb{A} \times \mathit{List\ Byte} \rfloor$ for these instructions. It also generates the proofs of the following properties asserting that the encoder and decoder are mutual inverses of each other. All the generated definitions and proofs are formalized in Coq.

**Theorem 1 (Consistency of Encoders and Decoders).**

$$\forall\, k\ l\ l', \mathbb{E}(k) = \lfloor l \rfloor \Longrightarrow \mathbb{D}(l{+}{+}l') = \lfloor (k, l') \rfloor.$$
$$\forall\, k\ l\ l', \mathbb{D}(l{+}{+}l') = \lfloor (k, l') \rfloor \Longrightarrow \mathbb{E}(k) = \lfloor l \rfloor.$$

```
token Opcode = (8);
token ModRM = (8);
field mod = ModRM(7:6);
field reg_op = ModRM(5:3);
field rm = ModRM(2:0);
token SIB = (8);
token Disp32 = (32);

token REX = (8);
field rmagic = REX(7:4);
field w = REX(3:3);
field r = REX(2:2);
field x = REX(1:1);
field b = REX(0:0);

class Addr =
  |constr addr_disp [Disp32]
  (mod=0x0 & rm=0x5; fld %1)
  ...
class Instruction =
  |constr mov_mr [reg_op, Addr]
    (Opcode=0x8B; fld %1 & cls %2)
  ...
```

**Fig. 3.** An Example of CSLED Specifications

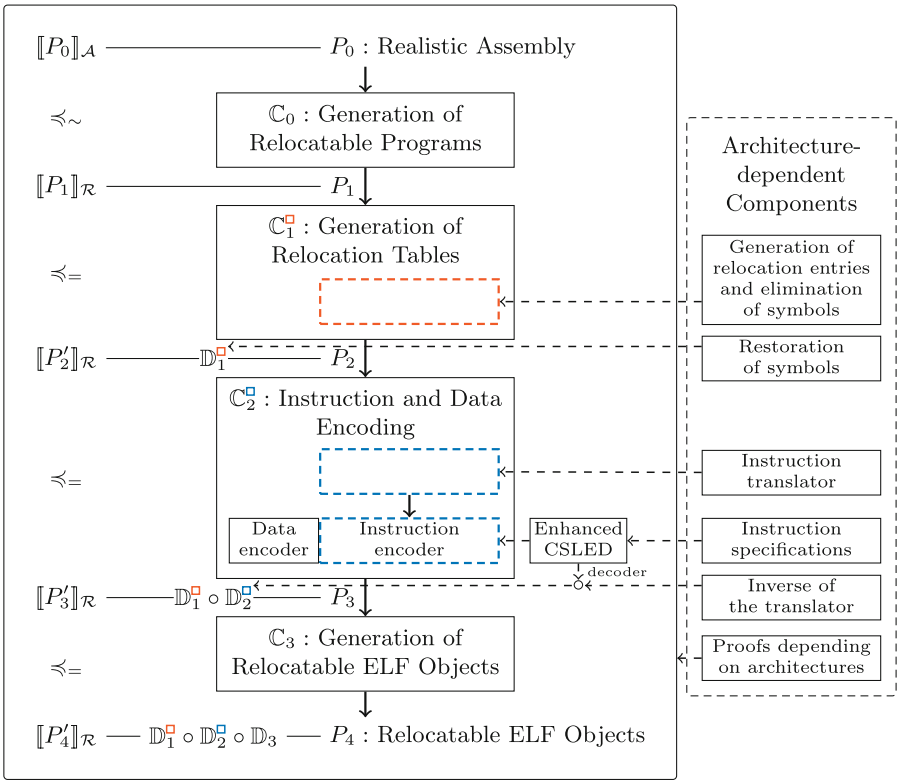
As an example, we show a snippet of the specifications of X86 instructions in Fig. 3. An instruction is built from *tokens*. Each token has one or more bytes (multiple of 8 bits). A *field* occupies a segment of a token, representing an operand or a constant. The tokens and fields reflect the instruction formats as described in Sect. 2.4. A *class* can be viewed as a variant whose binary form occupies a list of tokens. It is used to describe either a collection of instructions or its operands such as the addressing modes. Each branch of the `Instruction` class relates constant or values of operands to their corresponding fields in tokens or other classes through a *pattern* (written inside the parentheses). Here, the names of operands are listed in the brackets. The references to the $n$-th operand in the patterns are represented by `fld %n` or `cls %n`, depending on whether the operand is a field or a class. For example, the specification of `movl counter,%eax` in CSLED makes use of the branch with the constructor `mov_mr`. Its pattern corresponds to its encoding discussed in Sect. 2.4, such that the operand `reg_op` and the addressing mode are mapped into their corresponding binary tokens according to the pattern after the opcode `0x8B`.

## 3 The Framework

### 3.1 An Overview

Our framework is shown in Fig. 4. It can be viewed as a template of verified assemblers parameterized over architecture-dependent components, as depicted in the left box. This parameterization is achieved by exposing interfaces for encapsulation of architecture-dependent assembly processes. The main interfaces are highlighted with colored boxes in the assembly passes ($\mathbb{C}_1^{\square}$ and $\mathbb{C}_2^{\square}$) and disassembly functions ($\mathbb{D}_1^{\square}$ and $\mathbb{D}_2^{\square}$). Here, boxes with the same color represent interfaces for the same pass. The implementation of architecture-dependent components is shown in the right dashed box. By plugging them into the template through its interfaces, we get complete verified assemblers. The concrete definitions of these interfaces will be discussed in Sect. 3.3.

The main constituent of the template is a verified assembly chain with four passes, i.e., $\mathbb{C}_i (0 \leq i \leq 3)$. The source program is called Realistic Assembly or RealAsm in which every formalized assembly instruction corresponds to an actual machine instruction. The assembly chain transforms RealAsm programs into relocatable ELF objects through an intermediate representation called *relocatable programs* which is an abstract representation of ELF objects. We write




$\mathbb{C}_i$ : the $i$th assembly pass, where $\mathbb{C}_1^{\square}$ and $\mathbb{C}_2^{\square}$ expose interfaces
$P_i$ : the output of the $(i-1)$th pass
$\mathbb{D}_j$ : disassembly functions used to define semantics (discussed in Sec. 3.2) where $\mathbb{D}_1^{\square}$ and $\mathbb{D}_2^{\square}$ expose interfaces to be instantiated
$\mathcal{A}$ and $\mathcal{R}$ : realistic assembly languages and relocatable programs

**Fig. 4.** The Framework

$P_i (0 \leq i \leq 4)$ to represent these programs where $P_0$ is a RealAsm program, $P_4$ is an ELF object and the remaining ones are relocatable programs. Verification of the assembler is accomplished by proving lock-step forward simulation for every pass. To define the semantics for intermediate programs at different stages of assembly by reusing a single semantics of relocatable programs (denoted by $[\![\cdot]\!]_{\mathcal{R}}$), we define functions $\mathbb{D}_1^{\square}$, $\mathbb{D}_2^{\square}$, and $\mathbb{D}_3$ for reverting the assembly processes. The rationale for using such "disassembly" functions is given in Sect. 3.2. Another constituent of the template is the enhanced version of CSLED that supports both CISC and RISC instructions. It is used to automatically generate instruction encoders and decoders along with their consistency proofs from instruction specifications. In particular, the decoder is plugged into $\mathbb{D}_2^{\square}$ to implement the

inversion of instruction encoding. In the rest of this section, we elaborate on the representations and semantics of programs and the implementation of assembly passes along with their verification.

### 3.2 Source, Intermediate and Target Programs

**Memory Model.** CompCert's assembly language treats the stack as an unbounded linked list of stack frames, therefore requires pseudo instruction for stack manipulation (e.g., `Pallocframe` and `Pfreeframe`). To define semantics for realistic assembly programs without pseudo instructions, we adopt Stack-Aware CompCert's memory model for all the languages of our assembler. It enhances CompCert's memory model with a single and continuous stack [19], thereby enabling stack manipulation using the stack pointer instead of pseudo instructions.

**Realistic Assembly Programs.** A realistic assembly (or RealAsm) program is an instance of the assembly program of the type $P_{\mathcal{A}}$ introduced in Sect. 2.2, where the instructions (of type $I$) only contain real machine instructions. Its semantic is defined as an LTS consisting of *init* and *step* relations as introduced in Sect. 2.2. The initial memory as a result of calling *init* consists of a finite and continuous stack block, a unique block with initialized data for each internal function or variable, and a unique empty block for each external function or variable. The *step* relation of RealAsm programs is similar to CompCert's assembly except that no transition for pseudo instructions is defined and that it makes use of Stack-Aware CompCert's memory model.

**Relocatable Programs.** The relocatable program is a uniform intermediate representation for the assembly passes. It is a record parameterized by the instructions and data types ($I$ and $D$, respectively):

$$
\begin{aligned}
S &:= \lambda(I\ D : \mathit{Type}).\langle \mathit{code} : \mathit{List}\ I,\ \mathit{rwdata} : \mathit{List}\ D,\ \mathit{rodata} : \mathit{List}\ D\rangle \\
P_{\mathcal{R}} &:= \lambda(I\ D : \mathit{Type}).\{\mathit{sectbl} : \mathit{id} \to \lfloor S\ I\ D\rfloor,\ \mathit{symbtbl} : \mathit{id} \to \lfloor B\rfloor, \\
&\qquad\qquad \mathit{reloctbls} : \mathit{id} \to \lfloor \mathit{List}\ R\rfloor\}.
\end{aligned}
$$

Here, $P_{\mathcal{R}}$ encodes the four different kinds of ELF sections introduced in Sect. 2.3. It contains a table of sections (for code and data sections), a symbol table, and a mapping of relocation tables. They respectively map an identifier into a section (of type $S\ I\ D$), a symbol entry (of type $B$), and a relocation table (of type $\mathit{List}\ R$ where $R$ is the type of relocation entries). An element in the section table is either a code section containing a list of elements of type $I$, or a read-write or read-only data section containing elements of type $D$. The formal definitions of symbol entries and relocation entries mirror their informal definitions in Sect. 2.3.

The semantics for relocatable programs denoted as $[\![\cdot]\!]_{\mathcal{R}}$ serves as the uniform foundation for describing other languages' semantics (except for RealAsm)

in our framework. In this semantics, the order of memory blocks allocated during memory initialization is different from that for assembly or higher-level programs where memory blocks for global definitions are allocated in the same order as the definitions occurring in the program. In the definition of *init* in $[\![\cdot]\!]_{\mathcal{R}}$, the memory blocks for sections corresponding to internal definitions with non-empty initialization data or code are first allocated, then followed by the allocation of variable definitions with no initial values (corresponding to common symbols) and external definitions. The *step* relation is similar to RealAsm as it reuses the semantics of RealAsm's instructions. Note that $[\![\cdot]\!]_{\mathcal{R}}$ cannot be directly applied to $P_2$ and $P_3$ which, although also relocatable programs, are the results of further compilation by $\mathbb{C}_1^{\square}$ and $\mathbb{C}_2^{\square}$. To define their semantics by reusing $[\![\cdot]\!]_{\mathcal{R}}$, we first apply $\mathbb{D}_1^{\square}$ and $\mathbb{D}_2^{\square}$ to disassemble them and then apply $[\![\cdot]\!]_{\mathcal{R}}$. Therefore, their semantics are $[\![\mathbb{D}_1^{\square}(P_2)]\!]_{\mathcal{R}}$ and $[\![\mathbb{D}_1^{\square} \circ \mathbb{D}_2^{\square}(P_3)]\!]_{\mathcal{R}}$, respectively. The definitions of compilation and disassembly and their interfaces will be discussed in detail in Sect. 3.3.

**Relocatable ELF Objects.** Relocatable ELF objects (denoted by $\mathcal{E}$) formalize the ELF format introduced in Sect. 2.3. They are encoded as triples of the form $(E_h, E_s, E_{sh})$, where $E_h$ formalizes the ELF header, $E_s$ is a list of ELF sections in binary forms and $E_{sh}$ is a list of section headers. To define the semantics of a relocatable ELF program $P_4$ (denoted as $[\![P_4]\!]_{\mathcal{E}}$), we first use a function $\mathbb{D}_3$ which models ELF loading to get a relocatable program in binary form and then apply $\mathbb{D}_2^{\square}$ and $\mathbb{D}_1^{\square}$. That is, $[\![P_4]\!]_{\mathcal{E}}$ is formulated as $[\![\mathbb{D}_1^{\square} \circ \mathbb{D}_2^{\square} \circ \mathbb{D}_3(P_4)]\!]_{\mathcal{R}}$.

**Rationale Behind Disassembly.** As we have discussed above, we use disassembly functions to describe program semantics so that we only need a uniform semantics for relocatable programs which in turn reuses the semantics of assembly instructions. This greatly simplifies the verification of assemblers. This reliance on disassembly is not a fundamental limitation for two reasons. First, some form of disassembly is unavoidable for describing semantics for binary programs. For example, to describe the semantics of ELF, it is necessary to model ELF loading and instruction decoding, which are encoded in $\mathbb{D}_3$ and $\mathbb{D}_2^{\square}$ in our framework, respectively. Second, the structure of our framework does not change even if we use a more realistic ISA or ELF semantics without disassembly (e.g., Sail [2]). The only difference is that the forward simulation $\preccurlyeq_{=}$ need to be generalized to $\preccurlyeq_{\sim}$. Except for that, the structure of proofs should remain the same. Therefore, our framework is still applicable with more realistic binary semantics. The discussion about verification below should make these points clear.

### 3.3 Assembly Passes

The four assembly passes ($\mathbb{C}_0$ to $\mathbb{C}_3$) build relocatable ELF objects step-by-step. $\mathbb{C}_0$ and $\mathbb{C}_1^{\square}$ build the relocatable programs, among which $\mathbb{C}_0$ constructs a collection of sections and a symbol table from a RealAsm program and $\mathbb{C}_1^{\square}$ iterates the sections to generate relocation entries and eliminate unresolved symbols. $\mathbb{C}_2^{\square}$ performs instruction and data encoding that converts the contents in sections

into bytes. $\mathbb{C}_3$ generates relocatable ELF objects on a particular architecture (e.g., X86 or RISC-V). Their correctness are established as lock-step forward simulations as depicted in Fig. 4. The semantics of $P_0$ to $P_4$ have already been described in the last section. We use $\sim$ to denote the invariant for verifying $\mathbb{C}_0$ which relates the states of $P_{\mathcal{A}}$ and $P_{\mathcal{R}}$. For the remaining three passes, as the program semantics are defined by reverting the compilation, we use the equivalent relation $=$ as invariants. This in turn reduces lock-step simulation to proving correct that disassembly functions are exactly the inversion of compilation. Finally, by composing the correctness proofs of the four passes, we get the following semantics preservation theorem for our assembler:

**Theorem 2 (Semantics Preservation of the Assembler).**

$$\forall P\ P',\ \mathbb{C}_3 \circ \mathbb{C}_2^{\square} \circ \mathbb{C}_1^{\square} \circ \mathbb{C}_0(P) = \lfloor P' \rfloor \Longrightarrow [\![P]\!]_{\mathcal{A}} \preccurlyeq [\![P']\!]_{\mathcal{E}}.$$

In the remaining section, we discuss how to implement and verify these passes.

**Generation of Relocatable Programs.** This pass ($\mathbb{C}_0$) transforms a RealAsm program into a relocatable program containing sections and a symbol table in two steps. First, for every internal global definition in the source program, a corresponding section is built by invoking a function called *gen_section* to extract code or data from the definition. Second, a symbol table is created by repeatedly invoking *gen_symbol_entry* on *all* global definitions to get the symbol entries and inserting them into the initially empty symbol table. The types of $\mathbb{C}_0$, *gen_section* and *gen_symbol_entry* are given as follows:

$$\begin{aligned} \mathit{gen\_section} &: \forall I, (G\ (\mathit{Fd}\ I)\ \mathit{Unit}) \rightarrow \lfloor S\ I\ \mathit{Data} \rfloor \\ \mathit{gen\_symbol\_entry} &: \forall I, (G\ (\mathit{Fd}\ I)\ \mathit{Unit}) \rightarrow B \\ \mathbb{C}_0 &: \forall I, (P_{\mathcal{A}}\ I) \rightarrow \lfloor P_{\mathcal{R}}\ I\ \mathit{Data} \rfloor. \end{aligned}$$

Here, *Data* is the type of initial values of global variables defined in CompCert. For our running example, $\mathbb{C}_0$ generates two sections for `incr` and `main` and a symbol table with three symbol entries for the three global definitions. Therefore, the generated relocatable program mirrors the structure of the ELF object as depicted in Fig. 1c (except for the relocation tables). Note that the implementation of $\mathbb{C}_0$ is ignorant of $I$, therefore independent of architectures.

Given $\mathbb{C}_0(P_0) = \lfloor P_1 \rfloor$, we need to prove $[\![P_0]\!]_{\mathcal{A}} \preccurlyeq_{\sim} [\![P_1]\!]_{\mathcal{R}}$. Following the ideas described in Sect. 2.2, we define an invariant $\sim$ and prove that it holds for the initial states and is preserved by lock-step execution. The only non-trivial component of $\sim$ is a memory injection between source memory blocks for global definitions and corresponding target blocks for sections and symbols. The main difficulty of the proof is to show this injection indeed holds after initialization. Once the invariant is established, lock-step simulation naturally follows from it. Establishing this initial injection has been easy for all of CompCert's passes: since the global definitions for source and target are initialized in the same order, the injection is proved to hold by starting from an empty injection and incrementally showing that it is preserved after adding memory blocks for each pair

of corresponding source and target global definitions. However, this incremental approach no longer works for $\mathbb{C}_0$ because the order of initialization is changed. Consider our running example. In the source RealAsm program, the order of initialization is `counter`, `incr` and `main`. However, as described in Sect. 3.2, in relocatable programs memory blocks are first allocated for sections and then for the remaining symbols. As a result, the initialization order for the relocatable program of our example is `incr`, `main` and `counter`. Therefore, incremental pairing of definitions and growth of injection during initialization is no longer possible. To solve this problem, we directly prove that an injection between all source definitions and target blocks holds right after the initialization is completed. Because of its monolithic nature, this proof is considerably more complicated than the incremental proofs. Nevertheless, the initial injection can be directly established by observing that the source block initialized from a definition $g$ is related to the target block initialized from $gen\_section(g)$ or $gen\_symbol\_entry(g)$.

**Generation of Relocation Tables.** $\mathbb{C}_1^{\square}$ generates relocation entries for instructions or data that refer to symbols whose addresses are not determined until linking. For each code or data section, it generates one relocation table. To facilitate encoding of instructions and data into binary forms, it also eliminates the symbols in them. Its type is:

$$\mathbb{C}_1^{\square} : \forall I, (Z \to I \to (\lfloor R \rfloor \times I)) \to (P_{\mathcal{R}}\ I\ Data) \to \lfloor P_{\mathcal{R}}\ I\ Data \rfloor.$$

The first argument of $\mathbb{C}_1^{\square}$ is a parameter named $gen\_reloc$. As its color shows, it is part of the interfaces for encapsulating the architecture-dependent components. Given an instruction $i$ and its offset $o$ in $i$, $gen\_reloc\ o\ i$ produces a relocation entry for $i$ if $i$ contains a symbol and returns an updated instruction with the symbol replaced by the constant 0. For example, given `movl counter,%eax` and its offset inside the `incr` section, an instance of $gen\_reloc$ for the X86 architecture constructs a relocation entry for `counter` as described in Sect. 2.3. It also produces an updated instruction `movl 0,%eax` where `counter` is replaced by `0`. This makes the instruction independent of any symbol and hence can be encoded into bits.

Given $\mathbb{C}_1^{\square}(P_1) = \lfloor P_2 \rfloor$, we need to prove $[\![P_1]\!]_{\mathcal{R}} \preccurlyeq_{=} [\![\mathbb{D}_1^{\square}(P_2)]\!]_{\mathcal{R}}$. Note that, if we could show that $\mathbb{D}_1^{\square}$ reverts $\mathbb{C}_1^{\square}$, then the forward simulation holds trivially with an equality invariant. $\mathbb{D}_1^{\square}$ has the following type:

$$\mathbb{D}_1^{\square} : \forall I, (R \to I \to I) \to (P_{\mathcal{R}}\ I\ Data) \to (P_{\mathcal{R}}\ I\ Data).$$

Its first argument is called $restore\_symb$ and is also part of our framework's interfaces. Given an instruction $i$ and its relocation entry $r$, $restore\_symb\ r\ i$ extracts the symbol stored in $r$ and writes it back into $i$. For example, $restore\_symb$ converts `movl 0,%eax` back to `movl counter,%eax` given the generated relocation entry. The key to showing that $\mathbb{D}_1^{\square}$ reverts $\mathbb{C}_1^{\square}$ is to prove the following property, i.e., $restore\_symb$ reverts $gen\_reloc$, whose proof is straightforward:

$$\forall\ i\ i'\ e\ o,\ gen\_reloc\ o\ i = (\lfloor e \rfloor, i') \Longrightarrow restore\_symb\ e\ i' = i.$$

The above verification process is also applicable to the remaining two passes.

**Instruction and Data Encoding.** $\mathbb{C}_2^{\square}$ encodes instructions and data sections into sections containing bytes. It has the following type:

$$\mathbb{C}_2^{\square} : \forall I\ I', (I \rightarrow \lfloor List\ I' \rfloor) \rightarrow (I' \rightarrow \lfloor List\ Byte \rfloor) \rightarrow (P_{\mathcal{R}}\ I\ Data) \rightarrow \lfloor P_{\mathcal{R}}\ Byte\ Byte \rfloor.$$

The first two arguments are called *translate_instr* and *csled_encode*, respectively. They are also part of our framework's interfaces. *translate_instr* is a hand-written instruction translator for converting RealAsm instructions into a list of abstract assembly instructions characterized by the inductive definition for instructions generated from CSLED specifications (i.e., $\mathbb{A}$ introduced in Sect. 2.5). These CSLED instructions are subsequently encoded into bytes by *csled_encode* which makes use of the encoder generated from CSLED specifications (i.e., $\mathbb{E}$ in Sect. 2.5). Unlike instruction encoding, data encoding is independent of architectures. The data encoder (of type $Data \rightarrow List\ Byte$) is directly embedded into $\mathbb{C}_2^{\square}$. It encodes data of different types (e.g. int, float, or double) into bytes by using appropriate encoders for scalar values.

Given $\mathbb{C}_2^{\square}(P_2) = \lfloor P_3 \rfloor$, we need to prove $[\![\mathbb{D}_1^{\square}(P_2)]\!]_{\mathcal{R}} \preccurlyeq_{=} [\![\mathbb{D}_1^{\square} \circ \mathbb{D}_2^{\square}(P_3)]\!]_{\mathcal{R}}$. It follows by showing that $\mathbb{D}_2^{\square}$ reverts $\mathbb{C}_2^{\square}$. $\mathbb{D}_2^{\square}$ decodes binary instructions back to RealAsm instructions. It has the following type:

$$\mathbb{D}_2^{\square} : \forall\ I\ I', (List\ Byte \rightarrow \lfloor I' \rfloor) \rightarrow (List\ I' \rightarrow \lfloor I \rfloor) \rightarrow (P_{\mathcal{R}}\ Byte\ Byte) \rightarrow \lfloor P_{\mathcal{R}}\ I\ Byte \rfloor.$$

The first two arguments are called *csled_decode* and *revert_translate* where *csled_decode* is the instruction decoder generated by CSLED (i.e., $\mathbb{D}$ in Sect. 2.5) and *revert_translate* further decodes CSLED instructions into RealAsm assembly instructions. To show instruction encoding is reverted by $\mathbb{D}_2^{\square}$, the key is to prove that *revert_translate* reverts *translate_instr* and *csled_decode* reverts *csled_encode*. The former is easily proved manually with certain automation scripts in Coq. The latter follows directly from Theorem 1 which is automatically generated by CSLED. Note that there is no need to show data encoding can be reverted: we can prove that the initial memory values (in bytes) obtained from data of type *Data* are equal to those initialized from data of type *Byte*.

**Generation of Relocatable ELF Objects.** $\mathbb{C}_3 : (P_{\mathcal{R}}\ Byte\ Byte) \rightarrow P_{\mathcal{E}}$ encodes the symbol table and relocation tables to a list of ELF sections, and generates headers for all the sections. As mentioned in Sect. 3.2, the ELF semantics is defined by employing an ELF loader $\mathbb{D}_3$. To verify this pass, we show that $\mathbb{D}_3$ reverts $\mathbb{C}_3$. We elide a discussion of this proof as it is straightforward.

## 4 Applications

We demonstrate the effectiveness of our framework by building assemblers for X86 and RISC-V that support all the 32 and 64-bit X86 and RISC-V instructions used by CompCert. By design, all we need to do is to provide instances for the interfaces exposed by our framework.

```
class Instruction =
  | constr rex [w, r, x, b] (rmagic=0x4 & fld %1 & fld %2 & fld %3 & fld %4)
  | constr mov_mr [reg_op, Addr] (Opcode=0x8B; fld %1 & cls %2)
  |...
```

**Fig. 5.** A Snippet of the X86 Specifications

### 4.1 Building an Assembler for X86

To obtain instances of the interfaces for supporting X86 instructions in CompCert, the most challenging task is to write down the CSLED specifications that capture the complex X86 instruction format. Instantiation of the remaining interfaces (e.g., *gen_reloc* and *translate_instr*) is straightforward.

As demonstrated in Sect. 2.5, CSLED is already sufficient for specifying 32-bit X86 instructions. However, it is more difficult to support 64-bit X86 instructions which can be viewed as 32-bit instructions prepended with an REX prefix which extends operands to 64 bits. An obvious solution is to write down two versions of CSLED specifications: one for 32-bit without the REX prefix and the other for 64-bit with the prefix. However, this duplication is not only tedious and error-prone, but also generates inefficient encoders and decoders with bloated proofs. To solve this problem, we treat REX as a new "instruction", as depicted in Fig. 5 where the first operand (bit) `w` denotes whether the instruction is in 32-bit or 64-bit mode and the remaining three operands (bits) are used to encode the extended registers referred by the instruction. The key observation that enables the treatment of REX as a separate instruction is that, by the design of the X86 64-bit extension, the binary form of REX does not overlap with any regular instruction. Therefore, unambiguous encoders and decoders can be generated from the CSLED specifications in Fig. 5.

### 4.2 Building an Assembler for RISC-V

RISC architectures have more consistent and much simpler instruction formats than CISC architectures. For instance, no REX prefix is needed to distinguish between 32-bit and 64-bit instructions. Therefore, it is conceptually more straightforward to build assemblers for RISC-V than for X86. However, to apply our framework, we still need to address a practical problem: the original CSLED cannot directly support encoding and decoding of RISC-V instructions. The original CSLED describes an instruction as a sequence of bytes such that a field cannot span over more than one byte. Therefore, CSLED is insufficient for encoding many RISC-V instructions with this characteristic.

The root cause of the above problem is that the algorithm for generating encoders and decoders in CSLED treats *byte* as the atomic unit for binary data. To support RISC instructions, we switched the atomic unit to *bit* and refactored the algorithm so that it can still correctly generate encoders and decoders, and their correctness proofs. After that, it is easy to write a RISC-V instruction specification for CompCert and to generate a verified RISC-V assembler.


**Fig. 6.** The End-to-end Compilation Chain

# 5 Evaluation

## 5.1 Connecting with Stack-Aware CompCert

To evaluate the effectiveness of our approach, we connect our verified assemblers with Stack-Aware CompCert [19,21]. The complete compilation chain is shown in Fig. 6. The pretty printers translate CompCert assembly code into RealAsm code by expanding all the pseudo instructions into real assembly instructions. This phase is the only part not yet formally verified. The difficulty in its verification is mainly caused by the discrepancy between the memory models used by (Stack-Aware) CompCert and our verified assemblers. In particular, pointer values and certain scalar values stored in CompCert's memory are abstract and cannot be directly interpreted as binary values. As a result, the source and target semantics of the pretty printer cannot be matched via simulation. To solve this problem, we will need a version of Stack-Aware CompCert with a more concrete memory model. This is a non-trivial task and left for future work.

## 5.2 Statistics and Comparison

To examine the efficiency of our assemblers, we have applied our X86 and RISC-V compilation chains to the test suite provided by CompCert. Initially, we observed a 2.6% slowdown on average by running the code generated by our compilation chains and comparing it with the performance of the code generated by CompCert which uses the GNU assembler `as`. By inspecting the code generated by `as`, we discovered that it runs more efficiently by choosing instructions operating on aligned data (especially for floating-point values). We then modified our pretty printer to generate the same instructions, which brought the slowdown down to 1.1%. We conjecture that our performance can be further improved by choosing instructions with smaller immediate values (e.g., 8-bit instead of 32-bit), which may reduce cache misses. Such experiments are left to future work.

The statistics of our Coq development are shown in Table 1, where the numbers are measured in lines of code (LoC) and obtained by using `coqwc`. Note that we count Coq specifications and proofs separately. The framework column displays LoC for architecture-independent components, while the applications column displays LoC for architecture-dependent components. The second to fourth rows show the statistics for the program representations in our framework. The subsequent four rows are for the assembly passes. In instruction and data encoding, we show LoC for the manually written translators and the CSLED specifications separately. The next row shows the statistics for the pretty printers which are developed for each architecture (but not verified). As shown in the

**Table 1.** Statistics of Our Development

| Components | Framework | | Applications | | | |
|---|---|---|---|---|---|---|
| | | | X86 | | RISC-V | |
| | Spec | Proof | Spec | Proof | Spec | Proof |
| Realistic Assembly | 40 | 28 | 221 | 8 | 332 | 15 |
| Relocatable Programs | 1347 | 2165 | 797 | 48 | 423 | 38 |
| Relocatable ELF | 970 | 507 | 16 | 0 | 16 | 0 |
| Generation of Relocatable Programs | 685 | 1600 | 251 | 548 | 156 | 134 |
| Generation of Relocation Tables | 217 | 501 | 443 | 54 | 140 | 21 |
| Instruction and Data Encoding | 244 | 1016 | 0 | 0 | 0 | 0 |
| • Instruction Translators | 0 | 0 | 2178 | 469 | 2144 | 432 |
| • CSLED Specifications | 0 | 0 | 150 | 0 | 229 | 0 |
| Generation of Relocatable ELF | 605 | 1032 | 87 | 153 | 83 | 121 |
| Pretty Printers | 0 | 0 | 1005 | 0 | 1127 | 0 |
| Total | 4108 | 6849 | 5148 | 1280 | 4650 | 761 |

last row, a major part of the work for developing verified assemblers is isolated in the generic and architecture-independent framework. Note that a major part of the architecture-dependent development is for instruction translators (about 2.5k LoC each for X86 and RISC-V). However, we observe that these code and proofs are highly structured and may be simplified with further automation. The Coq proof scripts automatically generated by CSLED are quite large and may slow down the proof checking significantly when the number of instructions increases. We plan to solve this problem by dividing instructions into smaller categories which can be verified independently.

Finally, we compare our work with the most relevant existing work, i.e., CompCertELF [20] which also implements a verified assembler for the 32-bit X86 backend of CompCert. The main difference is that CompCertELF only supports a subset of X86-32 instructions and this support is hard-coded in its implementation. In particular, 89 X86-32 instructions of CompCert are implemented, out of which 24 are fully verified. This takes about 2300 LoC. As shown in Table 1 we only need 2647 LoC (2178 + 469) for the hand-written translator and 150 lines of the CSLED specifications to support the complete X86-32 and X86-64 backends of CompCert (a total of 146 instructions implemented and verified). Moreover, since CompCertELF does not separate the architecture-independent and -dependent implementation and proofs, it is unclear how it can be extended to support other architectures.

## 6 Related Work and Conclusion

*Verified Assembly.* To develop a verified assembler, it is necessary to precisely describe the semantics of assembly programs and object files. The semantics of

assembly programs in CompCert [8] is not ideal as it is not based on a realistic machine model (e.g., pointers can not be represented as binary values) [9]. There has been work on fixing these problems [3,4,13]. CompCertS [3] uses a concrete memory model to map memory blocks to 32-bit integers. Kang et al. [4] combines the logical and concrete memory models to enable injection (casting) of pointers into integers. Mullen et al. [13] defines a new semantics for X86-32 assembly which models pointers as 32-bit integers by introducing a memory allocator to translate memory blocks to concrete addresses. The memory model we use is based on Stack-Aware CompCert [19,21]. It extends the memory with an abstract stack to support the finite and continuous stacks in assembly programs.

There exists a lot of work on formalizing generation of low-level code (e.g., proof carrying code [1,14] and typed assembly [11,12]). However, none of them formally proves the correctness of assemblers. Recent work on verified compilation tried to address this problem. CakeML [6,16] is a verified compiler for a subset of Standard ML. Its backend supports compilation to machine code on different architectures. However, it uses an internal representation called `LABLANG` to store the encoded data instead of a standard binary file format [16]. CompCertELF [20] supports verified compilation from C programs all the way to the relocatable ELF files. However, it only supports a small subset of X86-32 and is difficult to extend due to its hard-coded dependency on X86-32. A translation validator known as *Valex* has been developed for the PowerPC assembler in CompCert [5]. It checks the consistency between generated executable programs and abstract assembly code. However, it is not formally verified.

*Instruction Encoding and Decoding.* CSLED [22] is a framework for automatically generating verified encoders and decoders from instruction specifications. Its specification language is based on the instruction specification language SLED [15] which does not provide any formal guarantee.

*Conclusion.* We have presented a framework for developing verified assemblers. It takes the form of a template implementing the architecture-independent parts of the verified assemblers. To obtain a verified assembler targeting a specific architecture, users only need to instantiate the architecture-dependent components exposed as interfaces in our framework. To demonstrate its effectiveness, we have applied our framework to develop assemblers sufficient to support the X86 and RISC-V backends of CompCert. We have further connected them with Stack-Aware CompCert via pretty printers and experimented on CompCert's official test suite. Our work is an initial attempt to develop realistic assemblers and linkers for end-to-end compiler verification. In the future, we would like to formally verify the pretty printers by using a more realistic memory model, extend our work to support verified linkers and verified compositional compilation, and scale our approach to other compilers, optimizations and object file formats.

**Acknowledgements.** We thank the anonymous referees for their feedback which improved this paper significantly. This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62002217.

## References

1. Appel, A.W.: Foundational proof-carrying code. In: Proceedings of 31st IEEE Symposium on Logic in Computer Science (LICS'16), pp. 247–256. IEEE Computer Society, Boston (2001). https://doi.org/10.1109/LICS.2001.932501
2. Armstrong, A., et al.: Isa semantics for armv8-a, risc-v, and cheri-mips. Proc. ACM Program. Lang. **3**(POPL), 71:1–71:31 (2019). https://doi.org/10.1145/3290384
3. Besson, F., Blazy, S., Wilke, P.: CompCertS: a memory-aware verified C compiler using pointer as integer semantics. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 81–97. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_6
4. Kang, J., Hur, C.K., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal c memory model supporting integer-pointer casts. In: Proceedings of 2015 ACM Conference on Programming Language Design and Implementation (PLDI 2015), pp. 326–335. ACM, New York (2015). https://doi.org/10.1145/2737924.2738005
5. Kästner, D., et al.: Compcert: practical experience on integrating and qualifying a formally verified optimizing compiler. In: Proceedings of 9th European Congress Embedded Real-Time Software and Systems, pp. 1–9. SEE (2018)
6. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. In: Proceedings of 41st ACM Symposium on Principles of Programming Languages (POPL 2014), pp. 179–191. ACM, New York (2014). https://doi.org/10.1145/2535838.2535841
7. Leroy, X.: The CompCert Verified Compiler (2005-2023). http://compcert.inria.fr/
8. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
9. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (2012). https://hal.inria.fr/hal-00703441
10. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformation. J. Autom. Reason. **41**(1), 1–31 (2008). https://doi.org/10.1007/s10817-008-9099-0
11. Morrisett, G., et al.: TALx86: a realistic typed assembly language. In: 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, pp. 25–35. Atlanta, GA, USA (1999)
12. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999). https://doi.org/10.1145/319301.319345
13. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for compcert. In: Proceedings of 2016 ACM Conference on Programming Language Design and Implementation (PLDI 2016). pp. 448–461. ACM, New York (2016). https://doi.org/10.1145/2980983.2908109
14. Necula, G.: Proof-carrying code. In: Proceedings of 24th ACM Symposium on Principles of Programming Languages (POPL 1997), pp. 106–119. ACM, New York (1997). https://doi.org/10.1145/263699.263712

15. Ramsey, N., Fernández, M.F.: Specifying representations of machine instructions. ACM Trans. Program. Lang. Syst. **19**(3), 492–524 (1997). https://doi.org/10.1145/256167.256225
16. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A.C.J., Owens, S., Norrish, M.: The verified cakeml compiler backend. J. Funct. Program. **29**, e2 (2019). https://doi.org/10.1017/S0956796818000229
17. The Coq development team: The Coq proof assistant (1999 - 2023). http://coq.inria.fr
18. The GNU development team: GNU Binutils (2000 - 2023). https://sourceware.org/binutils/
19. Wang, Y., Wilke, P., Shao, Z.: An abstract stack based approach to verified compositional compilation to machine code. Proc. ACM Program. Lang. **3**(POPL), 62:1–62:30 (2019). https://doi.org/10.1145/3290375
20. Wang, Y., Xu, X., Wilke, P., Shao, Z.: Compcertelf: verified separate compilation of c programs into elf object files. Proc. ACM Program. Lang. 4(OOPSLA) **197**, 1–197:28 (2020). https://doi.org/10.1145/3428265
21. Wang, Y., Zhang, L., Shao, Z., Koenig, J.: Verified compilation of C programs with a nominal memory model. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498686
22. Xu, X., Wu, J., Wang, Y., Yin, Z., Li, P.: Automatic generation and validation of instruction encoders and decoders. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 728–751. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_34