Automatic Generation and Validation of Instruction Encoders and Decoders

Xiangzhe Xu, Jinhua Wu, Yuting Wang(✉), Zhenguo Yin, and Pengfei Li

Shanghai Jiao Tong University, Shanghai 200240, China
yuting.wang@sjtu.edu.cn

Abstract. Verification of instruction encoders and decoders is essential for formalizing manipulation of machine code. The existing approaches cannot guarantee the critical *consistency* property, i.e., that an encoder and its corresponding decoder are mutual inverses of each other. We observe that consistent encoder-decoder pairs can be automatically derived from bijections inherently embedded in instruction formats. Based on this observation, we develop a framework for writing specifications that capture these bijections, for automatically generating encoders and decoders from these specifications, and for formally validating the consistency and soundness of the generated encoders and decoders by synthesizing proofs in Coq and discharging verification conditions using SMT solvers. We apply this framework to a subset of X86-32 instructions to illustrate its effectiveness in these regards. We also demonstrate that the generated encoders and decoders have reasonable performance.

Keywords: Formalized instruction formats · Verified parsing · Program synthesis · Proof synthesis · Translation validation

1 Introduction

Software that manipulates machine code such as compilers, OS kernels and binary analysis tools, relies on *instruction encoders and decoders* for extracting structural information of instructions from machine code and for translating such information back into binary forms. Because of the sheer amount of instructions provided by any instruction set architecture (ISA) and the complexity of instruction formats, it is extremely tedious and error-prone to implement instruction encoders and decoders by hand. Therefore, the literature contains abundant work on automatic generation of instruction encoders and decoders, often from specifications written in a formal language capable of concisely and accurately characterizing instruction formats on various ISAs [7,12,15].

Unfortunately, the above approaches generate little formal guarantee, therefore not suitable for rigorous analysis or verification of machine code. In those settings, instruction encoders and decoders are expected to be *consistent*, i.e., any encoder and its corresponding decoder are inverses of each other, and *sound*, i.e., they meet formal specifications of instruction formats that human could easily understand and check.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 728–751, 2021.
https://doi.org/10.1007/978-3-030-81688-9_34

Consistency is essential for verification of machine code because it guarantees that manipulation and reasoning over the abstract syntax of instructions can be mirrored precisely onto their binary forms. For example, verification of assemblers requires that instruction decoding reverts the assembling (encoding) process [20]. However, the previously proposed approaches to verifying instruction encoders and decoders all fail to establish consistency: to handle the complexity of instruction formats (especially that of CISC architectures), they employ expressive but ambiguous specifications such as context-free grammars or variants of regular expressions, from which it is impossible to derive consistent encoders and decoders. A representative example is the bidirectional grammar proposed by Tan and Morrisett [18]. It is an extension of regular expressions for writing instruction specifications from which verified encoders and decoders can be generated. However, because of the ambiguity of such specifications, two different abstract instructions may be encoded into the same *bit string* (i.e., a sequence of bits). When the decoder is deterministic, not all encoded instructions can be decoded back to the original instructions.

In this paper, we present an approach to automatic construction of instruction encoders and decoders that are verified to be consistent and sound. It is based on the observation that an instruction format inherently implies a bijection between abstract instructions and their binary forms that manifests as the determinacy of instruction decoding in actual hardware. This is true even for the most complicated CISC architectures. From a well-designed instruction specification that *precisely* captures this bijection, we are able to extract an appropriate representation of instructions, a pair of instruction encoder and decoder between this representation and the binary forms of instructions, and the consistency and soundness proofs of the encoder and decoder.

Based on the above ideas, we develop a framework for automatically generating consistent and sound instruction encoders and decoders. It extends the approach to specifying and generating instruction encoders and decoders proposed by Ramsey and Fernández [15] with mechanisms for *validating* their soundness and consistency by using theorem provers and SMT solvers. The framework consists of the following components (which are also our technical contributions):

- *A specification language for describing instruction formats.* This language is deliberately weaker in expressiveness than regular expressions while strong enough for describing instruction formats on common ISAs. Different from the existing ISA specification languages, it is rich enough for precisely capturing the syntactical structures of instructions and their operands, which implicitly encode a bijection between the abstract and the binary representations of instructions.
- *The algorithms for automatically generating encoders and decoders from instruction specifications.* Given any instruction specification, they generate an abstract syntax of instructions, a partial function from the abstract syntax to bit strings (i.e., an encoder) and a partial function from bit strings to the abstract syntax (i.e., a decoder). The generated definitions are formalized in

the Coq theorem prover so that the encoder and decoder can be formally validated later.
– *The algorithms for automatically validating the consistency and soundness of the generated encoders and decoders.* Given any instruction specification, they synthesize the consistency and soundness proofs for the generated encoder and decoder in Coq. This is possible because the bijection implied by the original specification guarantees that the encoder and decoder are inverses of each other, under the requirement that the binary "shapes" of different instructions or operands do not overlap with each other. This requirement is inherently satisfied by any instruction format, and can be easily proved with SMT solvers.

To demonstrate the effectiveness of our framework, we have applied it to a subset of 32-bit X86 instructions. In the rest of this paper, we first introduce relevant background information for this work and discuss the inadequacy of the existing work in Sect. 2. We then give an overview of our framework in Sect. 3 by further elaborating on the points above. After that, we discuss the definition of our specification language and the ideas supporting its design in Sect. 4. In the two subsequent sections Sect. 5 and Sect. 6, we discuss the algorithms for automatically generating and validating encoders and decoders. In Sect. 7, we present the evaluation of our framework. Finally, we discuss related work and conclude in Sect. 8.

2 Background

For our approach to work, the specification language we use must support the instruction formats on contemporary RISC and CISC architectures. In this section, we first introduce the key characteristics of these formats and then present a running example. We conclude this section by exposing the inadequacy of the existing approaches in capturing the bijections between the abstract and binary forms of instructions.

2.1 The Characteristics of Instruction Formats

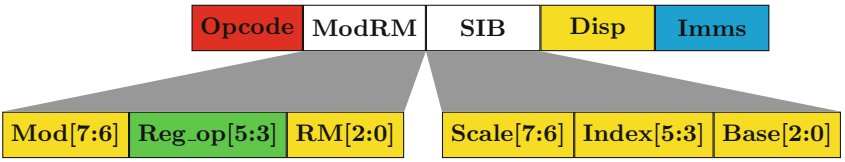


Fig. 1. The format of 32-bit X86 instructions

Instruction formats on CISC architectures may vary in length and structure even for the same type of instructions and may contain complex dependencies

between their operands. In contrast, instructions on RISC architectures usually have fixed formats which are largely subsumed by CISC formats. Therefore, we focus on handling CSIC formats in this paper.

We use the format of 32-bit X86 instructions as an example to illustrate the complex characteristics of CISC instructions. It is depicted in Fig. 1. An instruction is divided into a sequence of *tokens* where each token is one or more bytes playing a particular role. The first token **Opcode** partially or fully determines the basic type of the instruction; it may be one to three bytes long. Following **Opcode** is an one-byte token **ModRM**. **ModRM** is further divided into a sequence of *fields* where a field $f[n_1 : n_2]$ represents a segment of the token named f that occupies the n_2-th to n_1-th bits in that token. Depending on the value of **Opcode**, **ModRM** may or may not exist. When it exists, the value of **Reg_op[5:3]** may contain the encoded representation of a register operand. Another operand of the instruction may be an *addressing mode*. It is collectively determined by the values of **Mod[7:6]**, **RM[2:0]**, the token **SIB** (scaled index byte) and the displacement **Disp** following **ModRM**. Finally, the instruction may have an operand of immediate values in the token **Imms**.

For simplicity of our discussion, we have omitted some details such as the optional prefixes of instructions in Fig. 1. However, this simplified form is already enough to expose the key characteristics and complexity of CISC instruction formats (some of which also manifest in RISC). We summarize them below:

1. *Instructions as Composition of Components*: At the abstract level, an instruction consists of a collection of *components*. Each component serves a specific purpose and concretely corresponds to certain fields or tokens in the instruction format. For example, the constituents of 32-bit X86 instructions can be classified into four different kinds of components (marked with different colors in Fig. 1): the component determining the types of instructions (**Opcode**), the component denoting register operands (**Reg_op[5:3]**), the component denoting addressing modes (**Mod[7:6]**, **RM[2:0]**, **SIB** and **Disp**) and the component denoting immediate values (**Imms**).
2. *Variance of Components*: The concrete forms of components vary in different ways. A component may correspond to a single token (e.g., **Opcode** and **Imms**), a single field (e.g., **Reg_op[5:3]**), a mixing of fields and tokens (e.g., addressing modes), or other forms not shown here. Moreover, the abstract and concrete forms of a *single* type of components can also vary significantly such as the different addressing modes supported by X86 (as we shall see in detail in the following section).
3. *Interleaving of Components*. In most cases, there are clear sequential orders between the concrete representations of components. For example, the component of addressing modes immediately follows that of opcode and precedes that of immediate values. In the other cases, components may be interleaved with each other. For example, the component of register operands is interleaved with the component of addressing modes.
4. *Dependencies between and in Components*: The existence and forms of components are affected by the dependencies between each other and between their

own fields or tokens. For example, if an instruction does not take any argument, then the value of its **Opcode** determines that there is no token following **Opcode**. For another example, when **Mod[7:6]** contains the value `0b11`, the addressing mode is simply a register operand. Otherwise, the addressing mode may further depends on the values in **SIB** and **Disp**.

Note that, despite the above complexity, an instruction format is designed to inherently embed a (partial) bijection between the binary forms of instructions and their abstract representation as the composition of components. This is to ensure the determinacy of instruction decoding in hardware. This bijection is the central property to be investigated in this work.

2.2 A Running Example

Table 1. The different forms of addressing modes

AddrMode	Mod	RM	Scale	Index	Base	Disp
r	`0b11`	**r**	–	–	–	–
(**r**)	`0b00`	$\mathbf{r} \neq$ `0b100` $\wedge \mathbf{r} \neq$ `0b101`	–	–	–	–
(**d**)	`0b00`	`0b101`	–	–	–	**d**
$(\mathbf{s} * \mathbf{i} + \mathbf{b})$	`0b00`	`0b100`	**s**	$\mathbf{i} \neq$ `0b100`	$\mathbf{b} \neq$ `0b101`	–
...	...	...	...	...	...	...

We present an example of encoding the `add` instruction to concretely illustrate the characteristics of the X86 instruction format. It will be used as a running example for the rest of the paper. The operands of `add` may have many forms. For simplicity, we only consider two cases: *1)* the first operand is a register while the second one is an addressing mode, and *2)* the first operand is an addressing mode while the second one is an immediate value.

In the first case, **Opcode** is `0x03`, indicating that **ModRM** exists and the first operand is encoded in its **Reg_op** field. The addressing mode has over 23 combinations because of the dependencies and constraints over their fields. We list only some of the combinations in Table 1, where - indicates that this field or token does not exist. The first row shows the direct addressing mode **r** where **Mod** is `0b11` and **RM** contains the encoded register operand **r**. The following three rows shows different kinds of indirect addressing modes. They are valid only if **Mod** is `0b00` and further constraints are satisfied. For example, the second row shows the indirect addressing mode **(r)** where **r** is encoded in **RM**. In this case, **r** must neither be **ESP** (encoded as `0b100`) nor be **EBP** (encoded as `0b101`). Similarly, the addressing mode $(\mathbf{s} * \mathbf{i} + \mathbf{b})$ requires that **RM** must be `0b100`, **Index** must not be `0b100` and **Base** must not be `0b101`.

In the second case, **Opcode** is `0x81`, indicating that **ModRM** exists, the first operand is an addressing mode, and the second operand is an immediate value following it. Here, **Reg_Op** must be `0b000`.

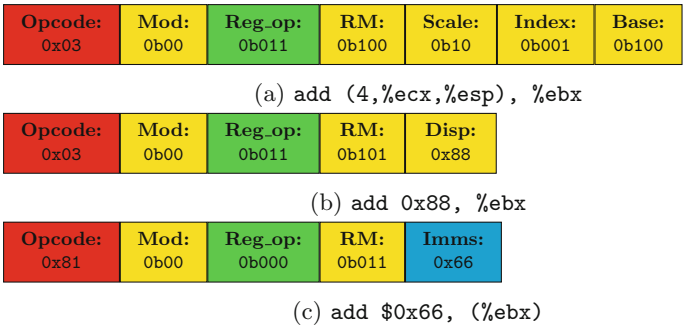


Fig. 2. Some concrete examples of instruction encoding

We demonstrate the concrete examples of encoding `add (4,%ecx,%esp), %ebx`, `add 0x88, %ebx` and `add $0x66, (%ebx)` in Fig. 2 where `%ebx` and `%ecx` are encoded into `0b011` and `0b001`, respectively (the order of operands is *reversed* because we use the AT&T assembly syntax). Note how the forms of operands change significantly depending on the different values in the related fields. Note also, despite such complex dependencies, a bit string representing a valid `add` instruction corresponds to a *unique* combination of components.

2.3 Inadequacy of the Existing Approaches

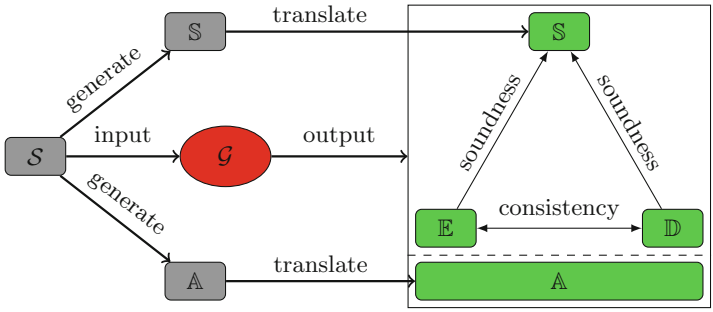
The existing approaches to specifying instructions are either *1)* too general and allow ambiguity or *2)* too low-level and break the component-based abstraction we just described. Either way, they fail to capture the inherent bijection embedded in an instruction format.

The bidirectional grammars [18] demonstrate the first kind of inadequacy. They contain the alternation grammar `Alt` g_1 g_2 for matching a bit string s when either the sub-grammar g_1 or g_2 matches s. The ambiguity arises when both g_1 and g_2 match s: in this case, the same s corresponds to two different internal representations. Therefore, bidirectional grammars cannot encode bijections in general. The same can be said for other work on verified parsing based on ambiguous grammars. We shall discuss them in detail in Sect. 8.

The Specification Language for Encoding and Decoding (or SLED) demonstrates the second kind of inadequacy [15]. It is a language for describing translations between symbolic and binary representations of machine instructions. On the surface, SLED takes the component-based view in specifying instructions. However, SLED specifications are interpreted through a normalization process by which every component is flattened into a sequence of tokens. After that, the structural information of components is completely lost. As a result, users can only derive encoders from the normalized specifications. They need to write decoders by using completely different specifications called "matching statements." This inability to generate matching encoders and decoders from a single specification is a common phenomenon in other approaches to ISA specifications.

In summary, no existing approach can precisely capture the bijections inherently embedded in instruction formats. This is the main intellectual problem we try to tackle in this paper. We shall elaborate on our solution to this problem in the remaining sections.

3 An Overview of the Framework



$\mathcal{S}$:	instruction specifications	(in CSLED)
$\mathcal{G}$:	algorithms for generating formal definitions and proofs	(in C++)
$\mathbb{A}$:	abstract syntax of instructions	(on paper and in Coq)
$\mathbb{S}$:	relational specifications of instructions	(on paper and in Coq)
$\mathbb{E}$ and $\mathbb{D}$:	encoders and decoders	(in Coq)

Fig. 3. The framework

We develop a framework for automatic generation of verified encoders and decoders that are consistent and sound. It is depicted in Fig. 3. To generate formally verified encoders and decoders, users first need to write down a specification of instructions $\mathcal{S}$ in a language called CSLED (or CoreSLED). CSLED is an enhancement to SLED for characterizing the bijection between the binary forms and the abstract syntax of instructions. Roughly speaking, $\mathcal{S}$ consists of a collection of *class* definitions, each of which defines a unique type of components that form instructions or their operands; the "top-most" class defines the type of instructions. Each class is associated with a set of *patterns* to uniquely determine a bijection between the binary and abstract forms of components in that class. Note that this bijection exists only when certain *well-formedness conditions* for patterns are satisfied. We shall elaborate on these ideas in Sect. 4.

From $\mathcal{S}$, the following definitions are generated and translated into Coq:

- The abstract syntax of instructions $\mathbb{A}$. It is a collection of algebraic data types corresponding to the classes defined in $\mathcal{S}$.
- A relational specification of $\mathcal{S}$ called $\mathbb{S}$. For each class, $\mathbb{S}$ contains a binary predicate that precisely captures the relation between components of that class and their binary forms. We write $\mathbb{R}[\![\mathcal{K}]\!]\ k\ l$ to denote that the component k of class $\mathcal{K}$ has the binary form l.

Then, $\mathcal{S}$ is fed into a collection of algorithms $\mathcal{G}$ to generate the following definitions and proofs in Coq:

- An encoder $\mathbb{E}$ and a decoder $\mathbb{D}$. The encoder is a set of partial functions—one for each class—from the abstract syntax of that class to bit strings. We write $\mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor$ to denote that l is the result of encoding a component k of class $\mathcal{K}$ where $\lfloor \rfloor$ denotes the `some` constructor of the option type. Conversely, the decoder is a set of partial functions from bit strings to the abstract syntax. We write $\mathbb{D}_{\mathcal{K}}(l{+}{+}l') = \lfloor (k, l') \rfloor$ to denote the decoding of the bit string l into a component k of class $\mathcal{K}$ where $+{+}$ is the append operation of bit strings. Here, the tailing bit string l' represents the remaining bits after decoding the first component.
- The proof of consistency between the encoder and decoder. The consistency theorems are stated as the mutual inversion between the encoder and decoder:

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

  Their Coq proofs are automatically generated by inspecting the logical structure of classes and patterns in $\mathcal{S}$. For this, we need to derive a very important property: the decoder always decodes a bit string l back to the same sequence of components. We achieve this goal by combining proofs in Coq with SMT solving of verification conditions that are automatically derived from well-formed specifications.
- The proof of soundness of the encoder and decoder. The soundness theorems are stated as follows:

$$\forall\, \mathcal{K}\ k\ l\ l', \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ l.$$
$$\forall\, \mathcal{K}\ k\ l\ l', \mathbb{D}_{\mathcal{K}}(l{+}{+}l') = \lfloor (k, l') \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ l.$$

  As we shall see later, $\mathbb{E}_{\mathcal{K}}$ and $\mathbb{R}[\![\mathcal{K}]\!]$ are both defined recursively on the definition of classes in $\mathcal{S}$. Their main difference is that the former is a function while the latter is a relation. Therefore, it is easy to prove the first soundness theorem by induction on k. By using the second consistency theorem and the first soundness theorem, we can easily prove the second soundness theorem.

As we shall see in the following sections, the actual implementations of encoders and decoders and their consistency and soundness theorems are more complicated than presented here. Nevertheless, the above discussion covers the high-level ideas of our framework.

Note that in Fig. 3, $\mathcal{S}$ and $\mathcal{G}$ are not formalized and hence not in the trusted base. The consistency and soundness of $\mathbb{E}$ and $\mathbb{D}$ are independently *validated* by using Coq and SMT solvers. If the validation of either property fails, the framework reports a failed attempt to generate the encoder and decoder. This often indicates that the instruction specification is not well-formed.

4 The Specification Language

The key idea underlying the design of CSLED is to record explicitly the structures of components in instruction specifications, instead of normalizing them into tokens as did in SLED. In this way, CSLED specifications accurately capture the key characteristics of instruction formats described in Sect. 2.1, hence the bijections embedded in instruction formats. In this section, we present the syntax of CSLED, explain the ideas underlying its design, and use the running example to illustrate how CSLED specifications are written. We also introduce the syntactical and relational interpretations of CSLED specifications and present the well-formedness conditions for the bijections to exist.

4.1 The Syntax

$$
\begin{array}{lll}
\mathcal{S} ::= \langle \texttt{empty} \rangle & & \mathcal{P} ::= \mathcal{J} \\
\quad\quad | \ \mathcal{S}\ \mathcal{D} & & \quad\quad | \ \mathcal{P}\,;\mathcal{J} \\
 & & \mathcal{J} ::= \mathcal{A} \\
\mathcal{D} ::= \texttt{token}\ \mathit{tid} = \mathcal{T}; & & \quad\quad | \ \mathcal{J}\,\&\mathcal{A} \\
\quad\quad | \ \texttt{field}\ \mathit{fid} = \mathcal{F}; & & \mathcal{A} ::= \mathcal{O} \\
\quad\quad | \ \texttt{class}\ \mathit{kid} = \mathcal{K}; & & \quad\quad | \ \texttt{cls}\ \%i \\
 & & \mathcal{O} ::= \epsilon : \mathit{tid} \\
\mathcal{T} ::= (n) & & \quad\quad | \ \mathit{fid} = n \\
\mathcal{F} ::= \mathit{tid}(n_1 : n_2) & & \quad\quad | \ \mathit{fid} \neq n \\
\mathcal{K} ::= \mathcal{B} & & \quad\quad | \ \texttt{fld}\ \%i \\
\quad\quad | \ \mathcal{K} \mid \mathcal{B} & & \quad\quad | \ \mathcal{O}\ \&\ \mathcal{O} \\
\mathcal{B} ::= \texttt{constr}\ \mathit{cid}\ [\mathit{aid}]\ (\mathcal{P}) & & \quad\quad | \ \mathcal{O}\ ;\ \mathcal{O}
\end{array}
$$

(a) Definitions (b) Patterns

Fig. 4. The syntax of CSLED

The syntax of CSLED is shown in Fig. 4. A CSLED specification (denoted by $\mathcal{S}$) consists of a list of *definitions* (denoted by $\mathcal{D}$). The three kinds of definitions are for tokens (denoted by $\mathcal{T}$), fields (denoted by $\mathcal{F}$) and classes (denoted by $\mathcal{K}$). Every definition is bound to a unique identifier where *tid*, *fid* and *kid* represents the identifiers of tokens, fields and classes, respectively.

Tokens represent consecutive segments of bytes and are the basic elements for forming instructions. They are necessary for distinguishing the same sequence of bytes with different interpretations. Their definitions have the form (n) where n must be divisible by 8 which denotes a token of n-bits or $n/8$ bytes. Definitions of fields have the form $\mathit{tid}(n_1 : n_2)$ which denotes a field occupying the n_2-th to n_1-th bits in the token *tid*.

Classes represent specific types of components. They play a central role in the specifications by accurately capturing the component-based abstraction we discussed in Sect. 2.1. A class consists of a collection of *branches* (denoted by $\mathcal{B}$) each of which denotes a possible form of components in the class. Definitions of branches have the form `constr` *cid* [*aid*] ($\mathcal{P}$) where *cid* is a unique identifier for the branch (denoting a constructor) and [*aid*] is a list of *fid* or *kid* denoting the sub-components or fields for constructing a component (i.e., the arguments to the constructor). These arguments capture the nested structures of components where a bigger component may be constructed from smaller ones or basic fields.

A branch is associated with a single *pattern* $\mathcal{P}$. A pattern plays two roles: it determines the types of a sequence of tokens that concretely forms components of this branch, and it describes a relation between these tokens (and their fields) with the abstract arguments of the branch. This relation essentially encodes the bijection between the abstract and binary forms of components in this branch.

At the top-most level, $\mathcal{P}$ is a sequence of *judgments* (denoted by $\mathcal{J}$) separated by `;`, such that $\mathcal{J}_1\,;\ldots;\mathcal{J}_n$ matches a sequence of tokens concretely represented by a bit string l if and only if $l = l_1{+}{+}l_2{+}{+}\ldots{+}{+}l_n$ and $\mathcal{J}_i$ matches l_i for $1 \leq i \leq n$. This sequential pattern is enough for relating abstract and binary forms of components when each $\mathcal{J}_i$ (and l_i) corresponds to a single (sub-)component. However, according to the discussion in Sect. 2.1, components may be interleaved with each other and $\mathcal{J}_i$ may correspond to multiple components. Therefore, a judgment is a conjunction of *atomic patterns* (denoted by $\mathcal{A}$) each of which matches an interleaved component. In case there is no interleaving, a judgment reduces to a single atomic pattern.

An atomic pattern has two forms: `cls` $\%i$ for relating a sequence of tokens to the i-th argument in [*aid*] of the corresponding branch which must be a class, and $\mathcal{O}$ for relating tokens to field arguments in [*aid*] and for further constraining the fields of these tokens. The $\mathcal{O}$ patterns are called *basic patterns*. Among them ϵ`:`*tid* matches any token of type *tid*; *fid* $= n$ (*fid* $\neq n$) matches a token with the field *fid* whose value is (is not) the constant n; similar to `cls` $\%i$, `fld` $\%i$ relates the i-th argument in [*aid*] of the branch which must be a field to the concrete value of the field in the matching token. The last two cases of basic patterns indicate that arbitrary sequencing and interleaving of basic patterns are allowed. Despite such free interleaving, a basic pattern can only match with sequences of tokens of the same length and of a unique type because we require that $\mathcal{O}_1$ `&` $\mathcal{O}_2$ be well-formed only if both $\mathcal{O}_1$ and $\mathcal{O}_2$ match sequences of tokens with the same type. Therefore, basic patterns have the same expressiveness as SLED specifications in their normalized forms [15].

In contrast to basic patterns, judgments and atomic patterns are much more expressive as they may match tokens of different lengths and forms. This is because a class pattern `cls` $\%i$ can match components of a class $\mathcal{K}$ with multiple branches, each of which may have different patterns. By introducing class patterns into atomic patterns, we are able to represent the complete structures of components and establish bijections from these structures. This is the key improvement we made in CSLED compared to SLED.

4.2 The CSLED Specification of the Running Example

```
token Opcode = (8);     token Disp = (32);     token Imms = (32);
token ModRM = (8);      token SIB = (8);

field opcode = Opcode(7 : 0);      field disp = Disp(31 : 0);
field imms = Imms(31 : 0);         field mod = ModRM(7 : 6);
field reg_op = ModRM(5 : 3);       field rm = ModRM(2 : 0);
field scale = SIB(7 : 6);          field index = SIB(5 : 3);
field base = SIB(2 : 0);

class Addrmode =
| constr addr_r [rm]  (mod = 0b11 & fld %1)
| constr addr_ir [rm]  (mod = 0b00 & rm ≠ 0b100 & rm ≠ 0b101 & fld %1)
| constr addr_disp [disp]  (mod = 0b00 & rm = 0b101; fld %1)
| constr addr_sib [scale, index, base]
   (mod = 0b00 & rm = 0b100;
   fld %1 & fld %2 & fld %3 & index ≠ 0b100 & base ≠ 0b101)
...

class Instruction =
| constr AddGvEv [reg_op, Addrmode]  (opcode = 0x03; fld %1 & cls %2)
| constr AddEvIz [Addrmode, imms]
   (opcode = 0x81; reg_op = 0b000 & cls %1; fld %2)
...
```

Fig. 5. The CSLED specification of the running example

The CSLED specification of our running example is depicted in Fig. 5. The *Addrmode* class specifies the possible addressing modes. Its branches are translated from the addressing modes described in Table 1 one by one, such that their patterns exactly match the binary structures of components in the corresponding branches. For instance, the branch *addr_sib* is translated from the fourth addressing mode in Table 1. Its pattern is a sequence of two judgment. The first judgment is a conjunction of two basic patterns that are the required constraints on the fields *mod* and *rm* of *ModRM* described in Table 1. Therefore, it must match the single token *ModRM*. The second judgment is a conjunction of basic patterns that constrain the fields *index* and *base* of *SIB* and relate arguments of *addr_sib* with the concrete values in the fields *scale*, *index* and *base*. Because these patterns all constrain the fields of *SIB*, the second judgment must match the single token *SIB*.

Similarly, the *Instruction* class specifies the instructions. Its two branches characterize the two kinds of `add` instructions described in Sect. 2.2. Note how conjunctions between the basic patterns for *reg_op* and class patterns for *Addrmode* are used to describe the interleaving of register operands and addressing modes. Note also that in every branch of *Addrmode* the first pattern matches

the token *ModRM*, and in any branch of *Instruction* the token *Opcode* is always followed by *Addrmode*. Therefore, *ModRM* always follows *Opcode* as desired.

By this example, we demonstrate the critical feature of CSLED: because the syntax of CSLED is designed to precisely describe instruction formats in ISA manuals, it implicitly captures the embedded bijections. Note that, because of its faithfulness to the ISA manuals, CSLED's syntax contains full details about instruction encoding by nature. However, it is not hard to imagine this syntax being refined to the client's syntax through another straightforward bijection. In fact, this is how we anticipate clients will use CSLED in practice, e.g., to build verified assemblers for X86.

4.3 Interpretation of CSLED Specifications

From a CSLED specification $\mathcal{S}$, we extract *1)* a collection of data types for representing the abstract syntax of components, and *2)* a collection of binary relations between these data types and bit strings for representing the mappings between the abstract and concrete forms of components.

Data Types of Components. We use the operator $\mathbb{T}[\![-]\!]$ to denote the interpretation of basic fields and classes into data types. The translation for fields are simple: given a field definition `field` $fid = tid(n_1 : n_2)$, $\mathbb{T}[\![fid]\!] = \langle n_1 - n_2 + 1 \rangle$ where $\langle n \rangle$ represent an unsigned binary integer of n bits. Note that we do not further translate the values of fields as they have straightforward interpretations (such as the mapping from bits to registers described in Sect. 2.1). The interpretation of classes is only slightly more involved. Given a class definition `class` $kid = \mathcal{K}$, $\mathbb{T}[\![kid]\!]$ is an algebraic data type named kid. For each branch `constr` $cid\ [aid_1, \ldots, aid_n]\ \mathcal{P}$ of $\mathcal{K}$, there is a constructor cid for kid that takes n arguments of types $\mathbb{T}[\![aid_1]\!], \ldots, \mathbb{T}[\![aid_n]\!]$.

Relations Derived from CSLED. The translation of CSLED specifications into relations is defined in Fig. 6. Here, BS denotes the type of bit strings. When $aids = [aid_1, \ldots, aid_n]$ we write $\mathbb{T}[\![aids]\!]$ to denote the product type of $\mathbb{T}[\![aid_1]\!], \ldots, \mathbb{T}[\![aid_n]\!]$. We use $\equiv$ to denote the definitional equality.

The function $\mathbb{R}[\![aid]\!]$ translates a type of components associated with aid into a binary relation between its abstract representation and bit strings, where aid may denote a field or a class. The definition for field components is straightforward. $\mathbb{R}[\![kid]\!]\ k\ l$ holds iff there is a branch of kid whose interpretation relates k and l, which further requires (by the third rule in Fig. 6) that k is constructed by using the constructor of that branch and the pattern of the branch relates the arguments of the constructor to l. The latter relation is defined by $\mathbb{R}_p[\![-,-]\!]$ such that $\mathbb{R}_p[\![\mathcal{P}, aids]\!]\ args\ l$ holds iff $\mathcal{P}$ matches l and the arguments $args$ satisfy the constraints enforced by $\mathcal{P}$ and $aids$. More specifically, $\mathbb{R}_p[\![\mathcal{P}\,;\mathcal{J}, aids]\!]\ args\ l$ holds iff $\mathcal{P}$ matches a prefix of l and $\mathcal{J}$ matches the rest of l. The definition of $\mathbb{R}_p[\![\mathcal{J}\,\&\mathcal{A}]\!]$ is slightly different in that $\mathbb{R}_p[\![\mathcal{J}\,\&\mathcal{A}, aids]\!]\ args\ l$ holds iff $\mathcal{A}$ matches the whole l and $\mathcal{J}$ matches a prefix of l. This is necessary for describing the

$$
\begin{aligned}
\mathbb{R}[\![\mathit{fid}]\!] &::= \lambda(f : \mathbb{T}[\![\mathit{fid}]\!])\ (l : BS).\\
&\quad \exists(\mathit{tid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv (n_3) \wedge \mathit{fid} \equiv \mathit{tid}(n_1 : n_2)\\
&\quad \wedge \mathit{length}(l) = n_3 \wedge l[n_1 : n_2] = f\\
\mathbb{R}[\![\mathit{kid}]\!] &::= \lambda(k : \mathbb{T}[\![\mathit{kid}]\!])\ (l : BS).\\
&\quad \exists \mathcal{B}, \mathit{kid} \equiv \ldots |\mathcal{B}| \ldots \wedge \mathbb{R}_b[\![\mathcal{B}, \mathit{kid}]\!]\ k\ l\\
\mathbb{R}_b[\![\mathcal{B}, \mathit{kid}]\!] &::= \lambda(k : \mathbb{T}[\![\mathit{kid}]\!])\ (l : BS).\\
&\quad \exists \mathit{args}, k = \mathit{cid}\ \mathit{args} \wedge \mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l\\
&\quad (\text{where } \mathcal{B} = \texttt{constr}\ \mathit{cid}\ \mathit{aids}\ \mathcal{P})\\
\mathbb{R}_p[\![\mathcal{P}\,;\mathcal{J}, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS).\exists l_1\ l_2, l = l_1 +\!\!+ l_2\\
&\quad \wedge \mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l_1 \wedge \mathbb{R}_p[\![\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l_2\\
\mathbb{R}_p[\![\mathcal{J}\,\&\mathcal{A}, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS).\exists l_1\ l_2, l = l_1 +\!\!+ l_2\\
&\quad \wedge \mathbb{R}_p[\![\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l_1 \wedge \mathbb{R}_p[\![\mathcal{A}, \mathit{aids}]\!]\ \mathit{args}\ l\\
\mathbb{R}_p[\![\epsilon : \mathit{tid}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS).\exists n, \mathit{tid} \equiv (n) \wedge \mathit{length}(l) = n\\
\mathbb{R}_p[\![\mathit{fid} = n, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS)\ .\exists(\mathit{tid}\ \mathit{fid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv (n_3)\\
&\quad \wedge \mathit{fid} \equiv \mathit{tid}(n_1 : n_2) \wedge \mathit{length}(l) = n_3 \wedge l[n_1 : n_2] = n\\
\mathbb{R}_p[\![\mathit{fid} \neq n, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS)\ .\exists(\mathit{tid}\ \mathit{fid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv (n_3)\\
&\quad \wedge \mathit{fid} \equiv \mathit{tid}(n_1 : n_2) \wedge \mathit{length}(l) = n_3 \wedge l[n_1 : n_2] \neq n\\
\mathbb{R}_p[\![\texttt{fld}\ \%i, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS).\mathbb{R}[\![\mathit{aids}[i]]\!]\ \mathit{args}[i]\ l\\
\mathbb{R}_p[\![\texttt{cls}\ \%i, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : BS).\mathbb{R}[\![\mathit{aids}[i]]\!]\ \mathit{args}[i]\ l
\end{aligned}
$$

Fig. 6. Translation of CSLED specifications into relations

interleaving of components. Furthermore, certain constraints need to be satisfied for deriving a bijection as shall discuss in Sect. 4.4. $\mathbb{R}_p[\![\mathcal{O}_1\,;\mathcal{O}_2, \mathit{aids}]\!]$ and $\mathbb{R}_p[\![\mathcal{O}_1\&\mathcal{O}_2, \mathit{aids}]\!]$ are not shown in Fig. 6 because they are defined the same as $\mathbb{R}_p[\![\mathcal{P}\,;\mathcal{J}, \mathit{aids}]\!]$ and $\mathbb{R}_p[\![\mathcal{J}\,\&\mathcal{A}, \mathit{aids}]\!]$, respectively. $\mathbb{R}_p[\![\mathit{fid} = n, \mathit{aids}]\!]$ *args* l holds iff l is a token containing *fid* whose value is n; similar for $\mathbb{R}_p[\![\mathit{fid} \neq n, \mathit{aids}]\!]$. $\mathbb{R}_p[\![\texttt{fld}\ \%i, \mathit{aids}]\!]$ holds iff the i-th argument in *args* matches with the concrete value found in l; same for $\mathbb{R}_p[\![\texttt{cls}\ \%i, \mathit{aids}]\!]$. Note how the last two definitions make use of *args* for getting the values of arguments.

4.4 Well-Formedness of Specifications

The binary relation we define in the last section denotes a bijection only when the CSLED specification under investigation satisfies certain well-formedness conditions. These conditions guarantee that, given any bit string l, there is at most one abstract object related to l via the defined binary relation. Well-formedness is the composition of three properties which we call *disjointness*, *compatibility*, and *uniqueness*. We give and explain their definitions below. The logic for checking these conditions is embedded in the generation algorithms we will discuss in the

next section and will be exploited for the validation of the generated encoders and decoders.

Disjointness. Given a pattern $\mathcal{P}_1\&\mathcal{P}_2$, it satisfies disjointness if $\mathcal{P}_1$ and $\mathcal{P}_2$ match disjoint fields.[1] To understand this, suppose $\mathcal{P}_1$ and $\mathcal{P}_2$ relate different abstract arguments a_1 and a_2 to overlapping bits in a bit string l. Then, we cannot determine if the values in the overlapping bits are for a_1 or a_2. Hence, the derived binary relation cannot possibly be a bijection. Disjointness rules out such possibility.

Compatibility. We call the types of sequences of tokens a pattern $\mathcal{P}$ matches the "shapes" of $\mathcal{P}$. Given a pattern $\mathcal{P}_1\&\mathcal{P}_2$, it satisfies compatibility if every possible shape of $\mathcal{P}_1$ is in a prefix of every possible shape of $\mathcal{P}_2$ when $\mathcal{P}_2$ is a class pattern (and vice versa). Enforcing compatibility simplifies the interpretation of $\mathcal{P}_1\&\mathcal{P}_2$ when $\mathcal{P}_1$ or $\mathcal{P}_2$ is a class pattern with multiple branches that may match bit strings with different shapes. Compatibility makes sense because for common instruction formats it is always the case that the components matched by $\mathcal{P}_1$ are embedded in the *longest common prefixes* of all the possible shapes of $\mathcal{P}_2$ when $\mathcal{P}_2$ is a class pattern (and vice versa). For example, in the example depicted in Fig. 2, **Reg_op** is always embedded into the common prefix of all the possible shapes of addressing modes, i.e., the **ModRM** token.

Uniqueness. Given a class pattern $\mathcal{K}$, it satisfies uniqueness if for any bit string l, at most one of its branches matches l. Uniqueness is essential for ensuring the determinacy of decoders in presences of class patterns. Fortunately, it implicitly holds for common instruction formats as they are designed with determinacy of decoding in mind. To concretely check the uniqueness implied by instruction formats, we first define the *structural condition* for a branch with pattern $\mathcal{P}$ as the conjunction of the statically known constraints in $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]_{cond}$. We then require that no structure conditions for any two branches of a class can be satisfied simultaneously. This requirement allows us to uniquely determine the branch used to construct a class component. For example, the structural conditions of the first three branches of *Addrmode* are $(mod = \texttt{0b11})$, $(mod = \texttt{0b00}\ \&\ rm \neq \texttt{0b100}\ \&\ rm \neq \texttt{0b101})$ and $(mod = \texttt{0b00}\ \&\ rm = \texttt{0b101})$. Obviously, any pairwise combination of these conditions cannot possibly be satisfied. This is true even if we consider all the branches of *Addrmode*. Therefore, there is at most one way to decode any addressing mode.

5 Generation of Encoders and Decoders

We discuss the algorithm for generating encoders and decoders from CSLED specifications. The structures of these encoders and decoders closely match the relations derived from specifications. Furthermore, every operation in an encoder has a counterpart in the corresponding decoder, and vice versa.

[1] We abuse the notation by using $\mathcal{P}$ to denote suitable patterns such as $\mathcal{J}$, $\mathcal{A}$ or $\mathcal{O}$.

5.1 Generation of Encoders

$$
\begin{aligned}
\mathcal{G}_{\mathbb{E}}[\![\epsilon : tid, bs, args]\!] &::= \lfloor bs \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![fid = n, bs, args]\!] &::= write_{fid}\ bs\ n \\
\mathcal{G}_{\mathbb{E}}[\![fid \neq n, bs, args]\!] &::= assert(read_{fid}\ bs \neq n) \\
\mathcal{G}_{\mathbb{E}}[\![\texttt{fld}\ \%i, bs, args]\!] &::= write_{fid}\ bs\ args[i] \qquad \text{(where } fid \text{ is the field id of } args[i]) \\
\mathcal{G}_{\mathbb{E}}[\![\texttt{cls}\ \%i, bs, args]\!] &::= \mathbb{E}_{\mathcal{K}}(args[i], bs) \qquad \text{(where } \mathcal{K} \text{ is the class of } args[i]) \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!] &::= l_1 \leftarrow first\_n(bs, [\![\mathcal{O}_1]\!]_{tokens}); l_2 \leftarrow skip\_n(bs, [\![\mathcal{O}_1]\!]_{tokens}) \\
&\quad\ bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1, l_1, args]\!]; bs_2 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_2, l_2, args]\!]; \lfloor bs_1 +\!\!+ bs_2 \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{\&}\ \mathcal{O}_2, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1, bs, args]\!]; \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_2, bs_1, args]\!] \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}\texttt{;}\mathcal{J}, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]; bs' \leftarrow skip\_n(bs, |bs_1|); \\
&\quad\ bs_2 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{J}, bs', args]\!]; \lfloor bs_1 +\!\!+ bs_2 \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{J}\texttt{\&}\mathcal{A}, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{J}, bs, args]\!]; \mathcal{G}_{\mathbb{E}}[\![\mathcal{A}, bs_1, args]\!]
\end{aligned}
$$

Fig. 7. Generation of encoders from patterns

From every class $\mathcal{K}$, we extract an encoder $\mathbb{E}_{\mathcal{K}}$ for its components. It is a partial function that takes two arguments—a component k and a bit string l representing the result previously generated by encoders—and outputs an updated bit string if the encoding succeeds. We shall write $\mathbb{E}_{\mathcal{K}}(k, l) = \lfloor l' \rfloor$ to denote that l' is the result of encoding k on top of l.

$\mathbb{E}_{\mathcal{K}}(k, l)$ is defined by recursion on the structure of k. For every branch $\mathcal{B}$ of $\mathcal{K}$, we generate a piece of Coq code from the pattern $\mathcal{P}$ of $\mathcal{B}$ for encoding k. We then insert it into the definition of $\mathbb{E}_{\mathcal{K}}(k, l)$. We write $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$ to denote the code snippet so generated, where bs is the name of the generated bit string at this point and $args$ contains the names of the arguments to the constructor. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$ is defined in Fig. 7 where we use the option monad for sequencing the encoding operations. The first case is obvious. Code generated by $\mathcal{G}_{\mathbb{E}}[\![fid = n, bs, args]\!]$ writes the constant n into the field associated with fid. $\mathcal{G}_{\mathbb{E}}[\![fid \neq n, bs, args]\!]$ checks whether the corresponding field contains the constant n and returns none if the checking fails. $\mathcal{G}_{\mathbb{E}}[\![\texttt{fld}\ \%i, bs, args]\!]$ writes the value of the i-th argument into the corresponding field. $\mathcal{G}_{\mathbb{E}}[\![\texttt{cls}\ \%i, bs, args]\!]$ calls the encoder for the class corresponding to $\texttt{cls}\ \%i$. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!]$ encodes its two parts recursively and concatenates the results together, where $first\_n(bs, n)$ returns the first n bits in bs and $skip\_n(bs, n)$ skips the first n bits in bs and returns the remaining ones. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\texttt{\&}\mathcal{O}_2, bs, args]\!]$ first encodes data matching $\mathcal{O}_1$, and then passes the result to the encoding for $\mathcal{O}_2$. The last two cases are similar. Note that if the generated code occurs at the beginning of a branch, then bs coincides with the input argument l. Otherwise, bs denotes intermediate results. As we can see, all these cases follow the logical structure of CLSED specifications we have described before.

5.2 Generation of Decoders

From every class $\mathcal{K}$, we extract a decoder $\mathbb{D}_{\mathcal{K}}$. It is a partial function such that $\mathbb{D}_{\mathcal{K}}(l) = \lfloor(k, l_1, l_2)\rfloor$ holds iff $l = l' {+\!+} l_2$, l' is the binary representation of k, and l_1 is the result of inverting the encoding operation, i.e., setting every bit the decoder touches in l' to 0. This extra return value is introduced to help with the verification as we shall see in Sect. 6.

$$
\begin{aligned}
\mathcal{G}_{\mathbb{D}}[\![\epsilon : tid, bs, args]\!] ::= {} & remains \leftarrow skip\_n(bs, tid); \lfloor(bs,\ remains)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![fid = n, bs, args]\!] ::= {} & ori \leftarrow clear_{fid}\ bs; remains \leftarrow skip\_n(bs, tid); \\
& \lfloor(ori,\ remains)\rfloor \qquad (\text{where } fid \equiv tid(n_1 : n_2)) \\
\mathcal{G}_{\mathbb{D}}[\![fid \neq n, bs, args]\!] ::= {} & ori \leftarrow clear_{fid}\ bs; remains \leftarrow skip\_n(bs, tid); \\
& \lfloor(ori,\ remains)\rfloor \qquad (\text{where } fid \equiv tid(n_1 : n_2)) \\
\mathcal{G}_{\mathbb{D}}[\![\texttt{fld}\ \%i, bs, args]\!] ::= {} & argi \leftarrow read_{fid}\ bs; ori \leftarrow clear_{fid}\ bs; \\
& remains \leftarrow skip\_n(bs, tid); \lfloor(ori,\ remains)\rfloor \\
& (\text{where } fid \text{ is the field id of } args[i]) \\
\mathcal{G}_{\mathbb{D}}[\![\texttt{cls}\ \%i, bs, args]\!] ::= {} & argi, origin, remains \leftarrow \mathbb{D}_{\mathcal{K}}(bs); \lfloor(origin,\ remains)\rfloor \\
& (\text{where } \mathcal{K} \text{ is the class of } args[i]) \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1\ ;\ \mathcal{O}_2, bs, args]\!] ::= {} & ori_1, remains_1 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1, bs, args]\!]; \\
& ori_2, remains_2 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_2, remains_1, args]\!]; \\
& \lfloor(ori_1 {+\!+} ori_2,\ remains_2)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1\ \texttt{\&}\ \mathcal{O}_2, bs, args]\!] ::= {} & remains \leftarrow skip\_n(bs, [\![\mathcal{O}_2]\!]_{tokens}); \\
& ori, \_ \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_2, bs, args]\!]; \\
& orilst, \_ \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1, ori, args]\!]; \\
& \lfloor(orilst,\ remains)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{P}\,;\mathcal{J}, bs, args]\!] ::= {} & ori_1, remains_1 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{P}, bs, args]\!]; \\
& ori_2, remains_2 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{J}, remains_1, args]\!]; \\
& \lfloor(ori_1 {+\!+} ori_2, remains_2)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{J}\,\&\mathcal{A}, bs, args]\!] ::= {} & ori, remains \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{A}, bs, args]\!]; \\
& orilst, \_ \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{J}, ori, args]\!]; \\
& \lfloor(orilst, remains)\rfloor
\end{aligned}
$$

Fig. 8. Generation of decoders from patterns

The first step of $\mathbb{D}_{\mathcal{K}}$ is to decide which branch of $\mathcal{K}$ should be chosen for decoding l. It can be done by checking the structural conditions derived from the patterns of branches (which we have introduced in Sect. 4.4) against l. Specifically, for the pattern $\mathcal{P}$ of each branch of $\mathcal{K}$, we translate its structural condition $[\![\mathcal{P}]\!]_{cond}$ into a decision procedure in Coq (a function returning boolean values) in a straightforward manner. We then insert an if-statement to check if $[\![\mathcal{P}]\!]_{cond}$ can be satisfied. If so, we start the decoding process for this branch. Otherwise, we

repeatedly check other branches until a matching case is found. Note also that by uniqueness, there is at most one structural condition that can be satisfied. Therefore, $\mathbb{D}_{\mathcal{K}}$ is deterministic in choosing branches.

Once a matching branch is found, we use the algorithm $\mathcal{G}_{\mathbb{D}}[\![\mathcal{P}, bs, args]\!]$ (the counterpart of $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$) to generate a piece of Coq code for decoding the arguments of this branch. It is defined in Fig. 8. Similar to encoding, the generated code snippet follows the logical structure of CSLED specifications. The function $clear_{fid}\ bs$ set the bits of the field *fid* in *bs* to 0. Note that the decoding operations are exactly the inversion of those in Fig. 7. Note also that the fourth and fifth cases in Fig. 8 are responsible for decoding the arguments and storing them in *argi*. By applying the corresponding constructor to these arguments, we get the output component k, which together with the two values returned by $\mathcal{G}_{\mathbb{D}}$ form the final output of $\mathbb{D}_{\mathcal{K}}$.

5.3 Generation for the Running Example

We show the representative cases of the generated encoder and decoder for our running example in Fig. 9. They include the encoding and decoding procedures for the fourth branch of *Addrmode* (the most complicated one). We can see that the encoding and decoding operations are exactly the inverses of each other. The encoder first writes the fields in *ModRM* and then those in *SIB*. Conversely, the decoder first reads the fields in *ModRM* and then those in *SIB*. Finally, it forms the component and returns the reverted and remaining bits. The function `BF_addr_sib` is the decision procedure generated from the structural condition for the fourth branch of *Addrmode*. We also show the encoding and decoding procedures for the first `add` instruction in Fig. 9. Their structures are very similar to those of *Addrmode*.

6 Validation of Encoders and Decoders

In this section, we discuss how to exploit the logical structure of and the well-formedness conditions for CSLED specifications to automatically synthesize the proofs of consistency and soundness for encoders and decoders.

6.1 Synthesizing the Proof of Consistency

The consistency between encoders and decoders is composed of two properties and stated as follows:

Theorem 1 (Consistency between Encoders and Decoders). *Given any class $\mathcal{K}$, its encoder $\mathbb{E}_{\mathcal{K}}$ and decoder $\mathbb{D}_{\mathcal{K}}$ are consistent with each other if they invert each other. That is, the following properties hold:*

$$\forall\ k\ l\ r\ l', valid\_input_{\mathcal{K}}(l) \Longrightarrow \mathbb{E}_{\mathcal{K}}(k, l) = \lfloor r \rfloor \Longrightarrow \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k, l, l') \rfloor.$$
$$\forall\ k\ l\ r\ l', \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k, l, l') \rfloor \Longrightarrow \mathbb{E}_{\mathcal{K}}(k, l) = \lfloor r \rfloor.$$

```
Definition encode_addrmode instance input :=
  match instance with
  ...
  | addr_sib arg1 arg2 arg3 ⇒
    (* Encode ModRM *)
    let ModRM := input in
    let tmp := write_mod ModRM b["00"] in
    let tmp := write_rm tmp b["100"] in
    let result0 := tmp in
    (* Encode SIB *)
    let SIB := zeros 8 in
    let tmp := write_scale SIB arg1 in
    let tmp := write_index tmp arg2 in
    let tmp := write_base tmp arg3 in
    let index := read_index tmp in
    let base := read_base tmp in
    do _ ← assert(index ≠ b["100"]);
    do _ ← assert(base ≠ b["101"]);
    let result1 := tmp in
    (* Concatenate the results of
       encoding ModRM and SIB *)
    Some (result0++result1)
  | ...
  end.
```

```
Definition decode_addrmode bs :=
  ...
  if BF_addr_sib bs then
    (* Revert the encoding of ModRM *)
    let ori := clear_mod bs in
    let ori := clear_rm ori in
    let ori1 := ori in
    do remains ← skipn bs 8; (* Skip ModRM *)
    (* Decode SIB to get the arguments
       and revert the encoding of SIB *)
    let bs := remains in
    let arg3 := read_base bs in
    let ori := clear_base bs in
    let arg2 := read_index ori in
    let ori := clear_index ori in
    let arg1 := read_scale ori in
    let ori := clear_scale ori in
    let ori2 := ori in
    do remains ← skipn bs 8;  (* Skip SIB *)
    (* Return the result *)
    Some(addr_sib arg1 arg2 arg3,
         ori1++ori2, remains)
  else if BF_addr_r bs then ...
    ...
```

```
Definition encode_instr instance input :=
  match instance with
  |AddGvEv arg1 arg2 ⇒
    ...
    let tmp := write_reg_op ModRM arg1 in
    do tmp ← encode_addrmode arg2 tmp;
    ...
  | ...
  end.

Definition decode_instr bs :=
  if BF_AddGvEv bs then
    ...
    do arg2, ori, remains ←
       decode_addrmode bs;
    let arg1 := read_reg_op ori in
    let ori := clear_reg_op ori in
    ...
```

```
Definition BF_addr_sib bs :=
  let ModRM := firstn bs 8 in
  (* mod = 0b00 ∧ rm = 0b100 *)
  let result0 :=
    (ModRM & b["11000111"]) = b["00000100"] in
  let tmp := skipn bs 8 in
  let SIB := firstn tmp 8 in
  (* index ≠ 0b100 *)
  let result10 :=
    (SIB & b["00111000"]) ≠ b["00100000"] in
  (* base ≠ 0b101 *)
  let result11 :=
    (SIB & b["00000111"]) ≠ b["00000101"] in
  result0 ∧ result10 ∧ result11.

Definition BF_AddGvEv bs :=
  let Opcode := firstn bs 8 in
  (Opcode & b["11111111"]) = b["00000011"].
```

Fig. 9. Encoders and decoders generated from the running example

We first discuss how the proof for the first property in Theorem 1 is generated. Here, the assumption $valid\_input_{\mathcal{K}}(l)$ asserts that all the bits in l that may be modified by $\mathbb{E}_{\mathcal{K}}$ must be 0. This is necessary to ensure that the decoder can revert the resulting bit string back to its initial state by setting them to 0 (i.e., the second result of decoding is the same as l).

The proof proceeds by induction on the structure of k. For each branch $\mathcal{B}$ with the pattern $\mathcal{P}$, we generate a lemma and its proof that the decision procedure generated from $[\![\mathcal{P}]\!]_{cond}$ as described in Sect. 5.2 always returns true given any bit string generated by the encoder for $\mathcal{P}$. With this lemma, the proof for the "symmetric" case where the decoder takes the same branch as the encoder reduces to proving that the encoder and decoder generated from $\mathcal{P}$ are inverses of each other. This proof is straightforward by the definitions of $\mathcal{G}_{\mathbb{E}}$ and $\mathcal{G}_{\mathbb{D}}$

in Sect. 5. An important point to note is that, for any pattern `cls` $\%i$, we need to recursively apply the consistency lemma for its corresponding class, which in turn requires us to establish a *valid_input* assumption. By the disjointness property in Sect. 4.4, we can easily conclude that the encoding of sub-components does not interfere with each other, thereby the desired *valid_input* assumption can be derived.

To finish the proof, we need to show that the "asymmetric" cases are not possible. For each asymmetric branch $\mathcal{B}'$ with the pattern $\mathcal{P}'$, we have that $[\![\mathcal{P}']\!]_{cond}$ holds by the decision procedure guarding this branch. Furthermore, by the above reasoning, $[\![\mathcal{P}]\!]_{cond}$ holds. We hence have that the conjunction of $[\![\mathcal{P}]\!]_{cond}$ and $[\![\mathcal{P}']\!]_{cond}$ holds. However, this contradicts with the uniqueness property given in Sect. 4.4. Therefore, the decoder can never go into a branch different from the encoder. Continue with our running example, suppose we are proving the consistency of the encoder and decoder for *Addrmode*. Further suppose we are working on the branch with the constructor *addr_sib*. Then, the verification condition for the asymmetric case with the constructor *addr_r* is

$$\forall bs, (read_{mod}\ bs = \texttt{0b00} \wedge read_{rm}\ bs = \texttt{0b100}\ldots) \wedge (read_{mod}\ bs = \texttt{0b11})$$

which cannot possibly hold (for simplicity we omit the conditions for *index* and *base*). We note that such condition can be easily checked by any SMT solver with the theory of bit-vectors, and we use Z3 [5] to validate them. This checking can also be directly formalized in Coq, which we plan to do in the future.

Finally, the second property in Theorem 1 can be proved by induction on k in a similar fashion. We elide a discussion of its proof.

6.2 Synthesizing the Proof of Soundness

As we have discussed in Sect. 4.3, the relational specifications extracted from CSLED specifications are tightly related to the actual instruction formats. Thus, it is reasonable to check the soundness of the generated encoders and decoders against these specifications. The relational specifications are easily translated into Coq definitions and we shall use the same notations. The soundness of encoders and decoders is then stated as follows:

Theorem 2 (Soundness of Encoders and Decoders). *Given any class $\mathcal{K}$, its encoder $\mathbb{E}_{\mathcal{K}}$ is sound if the following property holds:*

$$\forall\ k\ l\ r\ l', \mathbb{E}_{\mathcal{K}}(k, l) = \lfloor r \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ r.$$

Similarly, its encoder $\mathbb{D}_{\mathcal{K}}$ is sound if the following holds:

$$\forall\ k\ l\ r\ l', \mathbb{D}_{\mathcal{K}}(r{+\!+}l') = \lfloor (k, l, l') \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ r.$$

The soundness of encoder is easily proved by induction on the structure of k. We need to exploit the well-formedness conditions of CSLED specifications as for the consistency proofs at relevant points. The soundness of decoder is a corollary of the soundness of encoder and the second consistency property.

7 Evaluation

Besides the CSLED language, our framework has two major parts: *1)* the algorithms for generating encoders, decoders and their proofs and *2)* a Coq library containing the definitions and properties of basic types (including bits, bytes and bit strings) and a collection of automation tactics (Ltac definitions) for proof synthesis. The generation algorithms amount to 5,193 lines of C++ code (excluding comments and empty lines, and likewise for the following statistics). The Coq library amounts to 1,036 lines of Coq code (written in Coq 8.11.0 and counted using `coqwc`). We also make use of the monad definitions and some basic data formats in CompCert's library [13]. The whole framework took six person months to develop.

Table 2. The lines of generated Coq code

Component	Lines of definitions	Lines of proofs
Relational specification	1762	0
AST, encoder and decoder	5677	0
Verification conditions	37011	4402
Consistency proof	295	30841
Soundness proof	60	7193
Total	44805	42436

To evaluate the effectiveness of our framework, we have written a CSLED specification for a total of 186 representative X86-32 instructions which cover the operands with the most complicated formats (e.g., addressing modes) and are sufficient for supporting the assembling process in CompCert's X86-32 backend. The specification is very succinct, containing only 260 lines of CSLED code. From this specification, our framework *automatically* generates around 87k lines of Coq code which form the verified encoder and decoder. The lines of Coq definitions and proofs for individual components are shown in Table 2. Note that the verification conditions account for a major part of the definitions because we need to consider all the possible combinations of structural conditions for the proofs of consistency and soundness. The Coq proofs related to verification conditions are for identifying the concrete forms of structural conditions. As expected, the consistency proof is the most complicated one among all the proofs.

To evaluate the performance of the generated encoder and decoder, we randomly generate four sets of instructions, encode them into bit strings, and decode the bit strings back. The executable encoder and decoder are obtained by extracting Coq definitions into OCaml programs and compiling with OCaml 4.08.0. We repeat this experiment for 30 times on a machine with Intel(R) i7-4980HQ CPU@2.8 GHz and 16 GB memory. For comparison, we conduct the same experiments on the hand-written encoder and decoder in the X86-32 back-end of CompCertELF [20]. The results are shown in Table 3. For each test case, it shows the

Table 3. Performance evaluation

No. of Instr.	CSLED				Hand-Written			
	Enc. Time (s)		Dec. Time (s)		Enc. Time (s)		Dec. Time (s)	
	Med	Var.(%)	Med	Var.(%)	Med	Var.(%)	Med	Var.(%)
6000	0.32	0.00	0.56	0.00	0.01	0.00	0.01	0.00
12000	0.64	0.00	1.12	0.00	0.01	0.00	0.02	0.00
18000	0.98	0.03	1.70	0.15	0.02	0.00	0.03	0.01
60000	3.11	0.16	5.43	0.01	0.08	0.00	0.09	0.01

numbers of randomly generated instructions and the median time (in seconds) and the variance (in percentage) for encoding and decoding. We observe that the automatically generated encoder and decoder perform reasonably well, but significantly slower than the hand-written ones. This is because 1) the hand-written encoder and decoder in CompCertELF currently supports significantly less instructions (about 20) than the CLSED ones due to the complexity in manual implementation, and 2) the hand-written ones are manually optimized while the auto-generated ones are not optimized at all. We plan to solve the above issues by optimizing our generation algorithms in the future.

8 Related Work and Conclusion

We compare our framework with existing work on specification languages of instruction sets, verified parsing and pretty printing, and formalized ISAs.

There exists a lot of work on developing languages for specifying ISAs. Their major deficiency is the lack of formal guarantees. For example, the nML specification language employs attribute grammars to describe instruction sets [7]. For another example, EEL uses machine independent primitives to provide syntactic and semantic information of instructions [12]. The most relevant work in this category is the SLED language which our CSLED is based upon [15]. The patterns in SLED can only describe constraints on tokens and fields. By contrast, CSLED contains class patterns for accurately characterizing the structures of components. This extension enables CSLED to capture the bijection between the abstract and concrete forms of instructions.

Instruction decoding and encoding are special cases of parsing and pretty printing, respectively. Although there was early work on verifying that parsing and pretty-printing are inverses of each other by formulating them as bijections [1,10], this requirement was perceived as too strong [16]. Most of the recent work on verified parsing and pretty printing are dedicated to verify parser generators based on context-free grammars, regular expressions, parser combinators, or general data formats [3,11,17]. Some of them are also specialized work on verifying the encoder-decoder pairs [6,14,19,21]. They mostly deal with general and ambiguous grammars or specifications where bijection is difficult (if not impossible) to establish. By contrast, we intentionally restrict the expressiveness

of CSLED specifications to make proving consistency possible. Specifically, the syntax presented in Fig. 4 implies that CSLED specifications can only match sequences of tokens with finite lengths and shapes, making it strictly weaker than regular expressions, yet sufficiently strong for precisely capture the common instruction formats.

There is also abundant work on the development of formal ISA specifications (e.g., [2,4,8,9]). However, almost all of them focus on the problem of rigorously defining the *semantics* of ISAs (such as their sequential behaviors, concurrency models and interrupt behaviors). Although formalized encoders or decoders (or both) are sometimes generated (e.g., in Coq or Isabelle/HOL), there is no formal verification of the soundness or consistency of instruction encoding and decoding which only concerns the *syntax* of instructions.

In this paper, we have presented a framework for specifying instruction formats and for automatically generating and verifying encoders and decoders based on such specifications. The verified encoders and decoders are consistent with each other (being inverses of each other) and sound (conforming to high-level specifications). Consistency is provable in our framework because our specifications capture the bijections inherently embedded in instruction formats. In the future, we would like to apply this framework to a major part of X86-32 and X86-64 instructions and also to other ISAs, thereby to demonstrate the versatility and scalability of our framework.

Acknowledgments. We thank Zhong Shao for his comments and suggestions on this project when it was at an early stage. We are also indebted to anonymous reviewers for their detailed comments. This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62002217.

References

1. Alimarine, A., Smetsers, S., Weelden, A., Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, pp. 86–97. ACM (2005). https://doi.org/10.1145/1088348.1088357
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. Barthwal, A., Norrish, M.: Verified, executable parsing. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 160–174. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_12
4. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86–64 user-level instruction set architecture. In: PLDI 2019, pp. 1133–1148. ACM (2019). https://doi.org/10.1145/3314221.3314601
5. De Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

6. Delaware, B., Suriyakarn, S., Pit-Claudel, C., Ye, Q., Chlipala, A.: Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. Proc. ACM Program. Lang. **3**(ICFP), 82:1–82:29 (2019). https://doi.org/10.1145/3341686
7. Fauth, A., Van Praet, J., Freericks, M.: Describing instruction set processors using NML. In: Proceedings of the European Design and Test Conference, pp. 503–507. IEEE (1995). https://doi.org/10.1109/EDTC.1995.470354
8. Fox, A.: Directions in ISA specification. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 338–344. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_23
9. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the armv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_18
10. Jansson, P., Jeuring, J.: Polytypic compact printing and parsing. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 273–287. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49099-X_18
11. Jourdan, J.H., Pottier, F., Leroy, X.: Validating lr(1) parsers. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 397–416. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_20
12. Larus, J.R., Schnarr, E.: Eel: machine-independent executable editing. In: PLDI 1995, pp. 291–300. ACM (1995). https://doi.org/10.1145/207110.207163
13. Leroy, X.: The CompCert Verified Compiler (2005–2021). https://compcert.org/
14. Ramananandro, T., et al.: Everparse: verified secure zero-copy parsers for authenticated message formats. In: USENIX Security Symposium, pp. 1465–1482. USENIX Association (2019)
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. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. In: Proceedings of the 3rd ACM Haskell Symposium on Haskell, pp. 1–12. ACM (2010). https://doi.org/10.1145/1863523.1863525
17. Ridge, T.: Simple, functional, sound and complete parsing for all context-free grammars. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 103–118. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_10
18. Tan, Gang, Morrisett, Greg: Bidirectional grammars for machine-code decoding and encoding. J. Autom. Reason. **60**(3), 257–277 (2017). https://doi.org/10.1007/s10817-017-9429-1
19. Van Geest, M., Swierstra, W.: Generic packet descriptions: verified parsing and pretty printing of low-level data. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Type-Driven Development, pp. 30–40. ACM (2017). https://doi.org/10.1145/3122975.3122979
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. Ye, Q., Delaware, B.: A verified protocol buffer compiler. In: CPP 2019, pp. 222–233. ACM (2019). https://doi.org/10.1145/3293880.3294105