Automatic Checking of Instruction Specifications

Mary Fernández
AT&T Labs
600 Mountain Ave.
Murray Hill, NJ 07974
mff@research.att.com

Norman Ramsey
Dept. of Computer Science
University of Virginia
Charlottesville, VA, 22903
nr@cs.virginia.edu

ABSTRACT

Retargeting applications that process machine code is a tedious and error-prone task. We help automate retargeting by describing instruction sets in a high-level specification language and by generating automatically the code for encoding and decoding instructions. Moreover, we provide automated assistance for checking machine descriptions.

We check a specification for consistency both internally and externally. Internal checking forbids impossible instruction specifications, e.g., those that specify two values for a single bit. Internal checking warns about implausible specifications, e.g., those that don't specify values for all bits. Internal checks catch many mistakes, but they cannot guarantee that a specification accurately describes a target machine. Externally, we check a specification for consistency with an independent assembler. This technique increases a user's confidence in a specification's correctness and permits debugging in isolation of the target application. External validation is effective: it identified errors in specifications of the Intel Pentium, the MIPS, and the SPARC, *after* code generated from the latter two had been used in applications for over a year.

Keywords

Application generators. Machine-code toolkit. Specification testing.

INTRODUCTION

Many useful software-engineering tools can process machine-dependent information. Such tools include familiar testing tools, like Purify [10], which identifies hard-to-find memory leaks and errors. They also include profiling and tracing tools [2, 5], which help identify performance bottlenecks. Tools like these are most useful when they transform machine code, because machine-code tools can be applied to any executable program, even those for which source is unavailable. Processing machine code is also necessary for some less familiar, but increasingly important, tools. Run-time code generators, for example, are used in the implementations of functional and object-oriented languages [1, 9, 11], and they support rapid prototyping and late optimizations by delaying code generation until procedure-call time. Machine-level protection enforcers [24] improve software reliability by guaranteeing that mutually untrusted modules in a program cannot corrupt each other's data or code.

Working with machine code need not restrict a tool's generality, provided the tool can process or generate machine code on multiple architectures. Purify, for example, instruments executable programs for the SPARC, HP9000, and SGI/MIPS architectures. Support for multiple architectures comes at a cost; retargeting an application to a new architecture requires substantial effort. A significant part of this effort is implementation of instruction encoding and decoding. To implement encoding and decoding code by hand is error-prone, and to find the errors is tedious. These factors increase the costs to build and maintain retargetable applications—costs that are incurred for each new architecture. We have developed tools and techniques that reduce these costs and extend the lifetimes of machine-code tools by automating this part of the retargeting task.

We automate instruction encoding and decoding by describing the target instruction set at a high level and by generating automatically the code for bit manipulation. Programmers do not write low-level code to manipulate instructions; they use automatically generated code, which lets them encode and decode instructions at the same level of abstraction as assembly language. Programmers write and maintain high-level instruction specifications, not low-level, bit-manipulation code. Our small, high-level Specification Language for Encoding and Decoding (SLED) is suitable for describing the representations of instructions on both RISC and CISC architectures. We have written specifications for the MIPS R3000, SPARC, Intel Pentium, and Alpha; other programmers have written specifications for the Power PC and Motorola 68000. These architectures can be described with modest effort; a typical RISC specification is 100-200 lines, and the Pentium specification is about 500 lines. SLED is part of the New Jersey Machine-Code Toolkit. It is described in detail elsewhere [21].

Although SLED eliminates the need to write encoding and decoding code by hand, there remains the problem of accurately describing the representations of instructions on the target machine. The description takes the form of a SLED specification, which is more easily checked for correctness than the corresponding bit-manipulation code. We use both internal and external criteria to check specifications. Internally, the toolkit checks for both impossible and implausible instruction specifications. An impossible specification is meaningless and cannot be the description of any machine; for example, a specification that applies mutually unsatisfiable constraints to a range of bits is impossible. The toolkit rejects impossible specifications. An implausible specification is internally consistent, but is judged unlikely to describe a real machine. For example, it is implausible that the representation of an instruction would leave the values of some bits unspecified,[1] or that the representation of an instruction would not depend on the values of all the instruction's operands. The toolkit accepts implausible specifications, but it issues warning messages about the implausible aspects. Specifications with no impossible or implausible aspects are deemed plausible; they pass all internal checks.

The toolkit's internal checks catch many errors, but plausibility does not guarantee correctness. Common errors in plausible specifications include reversing the locations of an instruction's target and source registers, exchanging opcodes in a table, or simply mis-transcribing the value of an opcode. Errors like these can be detected by comparing representations of instructions as encoded by the toolkit with representations as encoded by some independent tool. The most natural tool is an assembler, which also converts symbolic to binary representations. Assemblers are available for all machines, and they are exercised by tools that use them frequently, like compilers and debuggers. If code generated by the toolkit produces a representation inconsistent with an assembler's representation, there is an error either in the SLED specification or in the assembler. Validation by comparison with assemblers effectively exposes errors in plausible SLED specifications; the technique described in this paper identified errors in our Pentium, MIPS, and SPARC specifications, and the MIPS and SPARC specifications had been used in an application for over a year without exposing these errors.

The overall contribution of our work is to to reduce retargeting costs, and therefore development and maintenance costs, by generating bit-manipulation code from a high-level specification. Our SLED specifications can be used to generate code for many different applications. Code generated from our specifications has been used in our own retargetable debugger [22] and retargetable, optimizing linker [6], and in a run-time code generator, a decompiler, and an execution-time analyzer [4], developed by others.

This paper describes the techniques we use to check a SLED specification for consistency with an independent assembler. By using such a widely available tool, we make it easy to check specifications. By exploiting the near-inverse relationship of assemblers and disassemblers, we make it easy to find the source of an inconsistency once it is identified. Finally, by exploiting knowledge about the structure of machine instructions, we exercise specifications thoroughly with a small number of tests. We also ensure that the tests expose such common errors as writing operands out of order and failing to identify operands that should be sign-extended.

[1]Exceptions exist. For example, the representation of the MIPS bc1f instruction leaves the value of bit 23 unspecified.

This paper is divided into two parts. Because our testing techniques are based on the structure of instructions as specified in SLED, we begin by describing SLED, and we show examples from a SLED specification of the SPARC. These examples, when concatenated, form a complete, valid specification of a subset of the SPARC. These examples also demonstrate the ease with which specifications can be written and maintained. For completeness, we also explain how SLED specifications are used, and how realistic applications can use the toolkit.

Readers already familiar with SLED or the toolkit can go directly to the second part of the paper, which shows how we use the information in the SLED specification to develop a good test sequence. The test-generation algorithm, which is the primary contribution of this paper, works independently of the particular application in which SLED might be used. It makes particular use of the SLED specification's *constructors*, which define assembly-language and binary representations of instructions. We describe how it checks for specific errors, such as sign-extending unsigned operands, and how it systematically tests a specification by checking at least one encoding of every addressing mode of every instruction. We describe the simple heuristics used to generate test values, and we show that these heuristics work well for instruction specifications. Finally, we describe how the specification checker and an assembler collaborate to validate a specification and how any assembler can be used as a validator, which makes our testing strategy feasible for any target.

SLED SPECIFICATIONS

SLED is suitable for describing the instruction sets of both CISC and RISC machines. Because uses of semantic information vary so widely, we believe it is useful to specify the representations of machine instructions independently of their semantics. By providing a high-level representation of instructions, and by automating the translation to and from that high-level representation, a SLED specification can support applications that have little in common semantically, like an assembler, a debugger, a linker, and a code generator.

To illustrate SLED, we describe a subset of the SPARC instruction set. The example, which is drawn from our complete, annotated specification of the SPARC [18], includes the SPARC's integer instructions, and it uses only those language features necessary to motivate the problem of checking specifications. The reader interested in more details or specifications of complex architectures can refer to a paper devoted to SLED [21] or to the toolkit's reference manual [19].

Because machine instructions do not always fit in a machine word, code generated by the toolkit works with *streams* of instruction *tokens*, not with individual instructions. A token stream is like a byte stream, except that the tokens may be integers of any size, not just 8-bit bytes. An instruction is a sequence of one or more tokens. Each token is partitioned into *fields*; a field is a contiguous range of bits within a token. Fields contain opcodes, operands, modes, or other information.

SLED contains two sublanguages: *patterns* describe binary representations, and *constructors* connect symbolic, assembly-language, and binary representations of instructions. Patterns constrain the values of fields; they may constrain fields in a single token or in a sequence of tokens. Uses vary; for example, simple patterns can be used to specify opcodes. More complex patterns can be used to specify addressing modes or to group instructions.

At a symbolic level, an instruction is a function (the constructor) applied to a list of operands. An operand may be as simple as a single field, or as complex as a set of fields taken from several tokens in sequence. Applying the constructor produces a pattern, which describes a sequence of tokens. Specification writers use constructors to define the equivalent of an assembly language. Application programmers use constructors to emit instructions, by calling procedures derived from constructor specifications, and to decode instructions, by using constructors in *matching statements* to match instructions and extract their operands. The specification checker uses constructors to emit both assembly-language and binary representations of instructions.

We have engineered the syntax of SLED to help writers of specifications maximize resemblances to architecture manuals. Our specification of the SPARC exploits this syntax; it is written to resemble parts of the SPARC architecture manual [23], and we refer to such parts by page number.

Architecture manuals usually have informal field specifications. In SLED, fields are specified by formal `fields` declarations. The following declaration defines the fields used in many SPARC instructions; it associates each field name with the range of bits in a token occupied by that field. The first two lines define the fields for some of the SPARC load instructions [23, p 90].

```
fields of itoken (32)
  op 30:31 rd 25:29 op3 19:24
  rs1 14:18 i 13:13 simm13 0:12
  op2 22:24 imm22 0:21 a 29:29 cond 25:28
  disp22 0:21 asi 5:12 rs2 0:4
```

Once we define fields, we can refer to them by name and do not have to refer to bit ranges.

Architecture manuals often define opcodes in tables. The SPARC manual uses a hierarchy of tables; we specify three. Tables F-1 and F-2 on p 227 are:

op[1:0]			
0	1	2	3
Table F-2	CALL	**Table F-3**	**Table F-4**

op2[2:0]				
0	2	4	6	7
UNIMP	Bicc **Table F-7**	SETHI NOP†	FBfcc **Table F-7**	CBccc **Table F-7**

In SLED, Tables F-1 and F-2 are specified by:

```
patterns
 [TABLE_F2 call TABLE_F3 TABLE_F4]
   is op = {0 to 3}
```

```
patterns
 [unimp _ Bicc _ sethi _ fbfcc cbccc ]
   is TABLE_F2 & op2 = {0 to 7}
```

The expressions in braces generate lists of patterns, and each pattern name in the bracketed list is bound to the corresponding pattern on the right. For example, `call` is bound to the pattern `op = 1`, and all opcodes in Table F-2 have `op = 0`, e.g., `Bicc` is bound to `op = 0 & op2 = 2`. Bindings to the wildcard "`_`" are ignored.

Table F-3 on p 228 defines opcodes for integer arithmetic; it is specified by:

```
patterns
   [ add    addcc    taddcc     wrxxx
     and    andcc    tsubcc     wrpsr
     or     orcc     taddcctv   wrwim
     xor    xorcc    tsubcctv   wrtbr
     sub    subcc    mulscc     fpop1
     andn   andncc   sll        fpop2
     orn    orncc    srl        cpop1
     xnor   xnorcc   sra        cpop2
     addx   addxcc   rdxxx      jmpl
     _      _        rdpsr      rett
     umul   umulcc   rdwim      ticc
     smul   smulcc   rdtbr      flush
     subx   subxcc   _          save
     _      _        _          restore
     udiv   udivcc   _          _
     sdiv   sdivcc   _          _      ]
 is
   TABLE_F3 & op3 = { 0 to 63 columns 4 }
```

The expression `{0 to 63 columns 4}` generates the integers from 0 to 63 in the sequence $(0, 16, 32, 48, 1, 17, 33, \ldots, 63)$, so that, for example, `addcc` is bound to `op = 2 & op3 = 16`, effectively using a column-major numbering.

Architecture manuals often group definitions of related instructions, such as all the integer-arithmetic instructions on pp 108–115. We use disjunctions of patterns to represent such groupings, which can make specifications more concise.

```
patterns
 arith is add  | addcc | addx   | addxcc | taddcc
        | sub  | subcc | subx   | subxcc | tsubcc
        | umul | smul  | umulcc | smulcc | mulscc
        | udiv | sdiv  | udivcc | sdivcc
        | save | restore | taddcctv | tsubcctv
```

Patterns allow us to refer to opcodes and other binary representations by name.

Most operands to instructions are simple fields or integers, but some operands have more structure. We use *typed* constructors to define such operands. The "effective address" operands, defined on p 84, have four possible formats:

```
constructors
dispA  rs1 + simm13!: addr is i = 1 & rs1 & simm13
absA   simm13!      : addr is i = 1 & rs1 = 0 & simm13
indexA rs1 + rs2    : addr is i = 0 & rs1 & rs2
indirA rs1          : addr is i = 0 & rs2 = 0 & rs1
```

Each line specifies a constructor by giving its opcode, operands, type, and pattern. The operand specification `simm13!` indicates a signed integer operand destined for field `simm13`.

The register or immediate operands (p 84) have two formats:

```
constructors
 rmode rs2     : reg_or_imm is i = 0 & rs2
 imode simm13! : reg_or_imm is i = 1 & simm13
```

We can use the type `reg_or_imm` to specify arithmetic instructions. Here is the definition of a group of untyped constructors, one for each of the 23 general-purpose arithmetic instructions:

```
constructors
  arith rs1, reg_or_imm, rd
```

This specification demonstrates two features of SLED, opcode expansion and implicit patterns. When the pattern `arith` is given as the opcode in a constructor specification, it is expanded into individual disjuncts, and the construct is equivalent to repeated specifications of `add`, `addcc`, `addx`, etc. The omission of the pattern indicates that the toolkit should compute a pattern by conjoining the opcode and all the operands, i.e., `add & rs1 & reg_or_imm & rd`. This idiom is common in specifications of RISC machines.

A constructor's operands may be decorated with spaces, commas, and other punctuation, which helps make its definition resemble the assembly-language syntax found in most manuals. This punctuation defines an assembly-language representation for the instruction, and the toolkit uses it to generate procedures that emit assembly language and to generate a grammar that recognizes assembly language.

The first column of Table F-7 on p 231 defines the branch opcodes. Although the target address is an operand to a branch, it is not represented directly as a field; instead, the target address is computed by adding a displacement to the program counter. We write an equation in curly braces to show the relationship, which is from pp 119–120:

```
relocatable target
constructors
  branch^a target { target = L + 4 * disp22! } is
    L: branch & disp22 & a
```

The label `L` refers to the location of the instruction, and the exclamation point is a sign-extension operator. The toolkit solves the equation so that the encoding procedure can compute `disp22` in terms of `target` and the program counter. This example designates `target` as *relocatable*, which means that its value may be unknown at encoding time. The most common example of such an instruction is a branch to an unknown label. Discussions of how the toolkit automates relocation of instructions appear elsewhere [16, 20].

Many assemblers extend the real instruction set with "synthetic" instructions. We specify synthetic instructions in terms of existing instructions by applying the corresponding constructors. Here is the definition of decrement, p 86:

```
constructors
 dec  val, rd  is  sub(rd, imode(val), rd)
```

Some synthetic instructions may stand for more than one instruction sequence, depending on the values of operands. We specify such instructions by putting alternatives on the right-hand side of a constructor specification, each with its own set of equations. The toolkit uses the first alternative, or *branch*, for which the equations have a solution. For example, the synthetic instruction `set` (p 84) expands to a single instruction when possible:

```
constructors
sethi val, rd is  sethi & rd & imm22 = val@[10:31]
set val, rd
 when { val@[0:9] = 0 } is sethi(val, rd)
 otherwise              is or(0, imode(val), rd)
 otherwise is sethi(val, rd);
              or(rd, imode(val@[0:9]), rd)
```

The bit-extraction operator, `@[low:high]`, extracts the bits in the positions from `low` to `high`. The first branch, `sethi`, can be encoded whenever the least-significant 10 bits of `val` are zero. The second branch works when `imode(val)` can be encoded, i.e., when `val` fits in 13 signed bits. The final branch can always be encoded. To test this kind of instruction, the toolkit's checker needs to find operands that exercise each of the constructor's three branches. For example, to exercise the second branch, it must find a value for `val` that fits into 13 signed bits, but in which one of the least-significant 10 bits is nonzero.

Using the Toolkit in Applications

Applications use the toolkit for encoding, decoding, or both. For example, assemblers encode, disassemblers decode, and some profilers do both. Decoding applications use *matching statements* to read instructions from an instruction stream and identify them. Matching statements make an application's decoding logic clear and concise. A matching statement is like a case statement, except its alternatives are labeled with patterns that match instructions or sequences of instructions. The matching statement in Figure 1 finds which instructions could be executed immediately after an instruction at which a breakpoint has been placed.[2] It is used to implement control-flow analysis in a retargetable debugger for ANSI C. This matching statement expands to nested case statements totaling about 90 lines of Modula-3 code, but the count does not convey the difficulty of writing the code by hand, because the toolkit combines seemingly unrelated opcodes if they result in executing the same arm of the matching statement.

Encoding applications call C procedures generated by the toolkit. These procedures encode instructions and emit them into an instruction stream; e.g., the SPARC call `add(r2, rmode(r3), r7)` emits the word `0x8e008003`.

The toolkit uses specifications in two ways. It can take a specification and generate encoding and relocation procedures in C for each constructor in the specification. It can also take a specification and a program with embedded matching statements and translate these statements into ordinary code, which can match the instructions or parts

[2] `epsilon` matches the empty pattern.

```
PROCEDURE Follow(m:Memory.T; pc:Word.T):FollowSet.T =
BEGIN
  match pc to
  | nonbranch; L: epsilon              => RETURN FollowSet.T{L};
  | call(target)                       => RETURN FollowSet.T{target};
  | branch^a(target) & (ba | fba | cba) => RETURN FollowSet.T{target};
  | branch^a(target); L: epsilon       => RETURN FollowSet.T{L, target};
  | jmpl(dispA(rs1, simm13), rd)       => RETURN FollowSet.T{GetReg(m, rs1)+simm13};
  | jmpl(indexA(rs1, rs2), rd)         => RETURN FollowSet.T{GetReg(m, rs1)+GetReg(m, rs2)};
  | some itoken                        => Error.Fail("unrecognized instruction");
  endmatch
END Follow;
```

Figure 1: Matching statement used for control-flow analysis of SPARC instructions

thereof defined in the specification. The toolkit can handle programs written in C or Modula-3 [13].

The toolkit ensures consistency of an instruction's encoding and decoding by deriving both types of code from the same part of the SLED specification. Each constructor specification produces a set of equations that relate the constructor's operands to the fields in its binary representation. To encode, the toolkit's equation solver takes operands as known and solves for fields. To decode, the solver takes fields as known and solves for operands. The solver does not compute numerical solutions directly; it computes symbolic solutions, which are used to generate C or Modula–3 code [17]. The generated code does the arithmetic when the application is run and checks that all conditions are satisfied for the constructor to be matched or encoded.

The checker uses the equation solver to help automate testing, so it is worth describing the form of the equations used. Explicit equations relate sums of terms with integer coefficients. They include not only equalities, but also inequalities, which are useful for expressing constraints on machines like the Pentium and Motorola 68000. Terms include field and integer variables, which can be sign-extended (widened). They can also have bits extracted, and those bits can be sign-extended. The equation solver introduces other operations, including narrowing and integer division and modulus. When necessary, we denote widening by $n\uparrow_k$, which takes the low-order k bits of a two's-complement integer n and considers those k bits as a two's-complement integer. Narrowing, $n\downarrow_k$, is its inverse operation. Narrowing succeeds only if it can be done without loss of information. Bit extraction, or $n[i{:}j]$, considers bits i through j of n as an unsigned integer, and it always succeeds.

Most equations are implicit. For example, the SPARC `imode` constructor implicitly equates the integer operand *simm13*, narrowed to 13 bits, to the field `simm13`, yielding the equation $\mathit{simm13}\downarrow_{13} = \texttt{simm13}$. The equation solver may also produce implicit *conditions* that must hold if the equations are to have a solution. For example, the equations for SPARC branch constructors are solvable if and only if `(target - L) mod 4 = 0`, because the SPARC branch format can only represent branches in which the target address and the program counter differ by a multiple of 4.

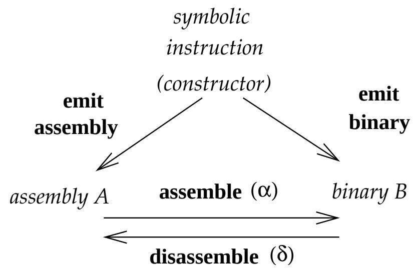


Figure 2: Isomorphism between assembly and binary

CHECKING FOR CORRECTNESS

A SLED specification provides mappings between symbolic, assembly, and binary forms of instructions. This mapping is similar to that provided by an assembler and disassembler. The mappings differ in details, e.g., the toolkit's symbolic instructions are represented by procedures, whereas assembly and binary instructions are represented by text and binary tokens. Figure 2 shows the isomorphism between mappings: all paths from an instruction to an assembly or binary representation should produce identical representations of that instruction. Given this property, we can check a specification as follows:

1. Choose a sequence S of instructions to test.
2. Use the toolkit to encode S in assembly file A and in binary file B.
3. Apply an independent assembler (α) to A to produce B', or apply an independent disassembler (δ) to B to produce A'.
4. Compare B with B' or A with A'

If the comparison shows differences, there is an inconsistency between the assembler's and the specification's encoding of S. There are a few tricky details involved in comparing results of assembly or disassembly, but the real difficulty lies in finding a sequence S of valid instructions that provide good coverage of the machine's instruction set.

Using an assembler to validate a specification is not foolproof, because it assumes that all instructions are used often enough to validate the assembler. Rarely used instructions may conceal bugs. For example, the C-30 IMP was designed with both left- and right-shift instructions, only one of which was used in practice. The first use of the unused shift revealed that the microcode was wrong, but the problem went undetected for several years.[3] Similar undetected bugs can exist in assemblers; in these cases, our technique would debug the assembler rather than the specification.

Even if the validating assembler is flawed, our technique identifies encoding inconsistencies, whether caused by an error in a specification, the assembler, or both. Moreover, errors are identified *before* toolkit-generated code is integrated into a large application, which simplifies the user's task of finding the error's source in a specification and/or the assembler. Our technique, however, does not handle the possibility that the specification and an independent assembler contain the same error. Such a case might arise if both the author of the specification and the author of the assembler misinterpreted the architecture manual in the same way.

Selecting Instructions and Operand Values

The selection of instructions and of operand values that exercise those instructions is the core of our testing method. We want the sequence of test instructions (S) to provide "good coverage" of a specification. For example, S should include at least one encoding of every branch of every constructor and one encoding of every instance of a constructor-typed operand, e.g., every format of an effective address.

Checking every encoding of every instruction is impractical, because there are too many total encodings. For example, there are almost 2^{31} valid encodings of SPARC instructions. Even though more than half of all possible encodings are invalid, exhaustively checking close to 2 billion valid encodings is infeasible. There are three sources of encoding variations: the values of integer and field operands, the multiple representations of complex operands, like effective addresses, and the multiple encodings for an instruction that are introduced by branches. For example, the `reg_or_imm` operand has at least two encodings, one for each of its instances, `rmode` and `imode`, and the `set` instruction has at least three encodings, one for each of the three branches guarded by conditions.

Variations in operands' values produce almost all of the encodings, so it is practical to select a tuple of operand values that is valid for the instruction and to check one encoding using those values. If an instruction has n operands, $V = \{(v_1, \ldots, v_n)\}$ is a set containing one tuple, where v_i is the test value of the i-th operand. For instructions without equations, selecting valid values for fields and integers is simple: select v_i to be a random value in the operand's range. For example, to test the SPARC `add` instruction, the checker selects random values in $\{0 \ldots 31\}$ for the operands `rs1` and `rd`, because both fields must fit in 5 bits. We detect out-of-order operands by ensuring that all operand values are distinct, and we detect sign-extension errors by randomly setting the sign bit in integer operands. Failure to sign-extend a signed operand is detected, because the independent assembler rejects a large, unsigned value that exceeds the signed operand's limits; similarly, sign extension of an unsigned operand is detected, because the assembler rejects negative values for unsigned operands.

[3] We thank the third reviewer for calling the C-30 IMP to our attention.

The checker exercises addressing modes, or register or immediate operands, by using different constructors to create operands of the appropriate constructor type. For each operand of a constructor type, the checker repeats the value-selection process recursively on constructors of that type, producing a set of tuple values for the operand. For the `add` instruction, for example, the checker selects test values for each constructor of the `reg_or_imm` type. For the `rmode` constructor, it selects a random 5-bit value for `rs2` and for the `imode` constructor, it selects a random 13-bit, signed value for `simm13`. The checker enumerates all combinations of values for the integer, field, and constructor-typed operands of an instruction and produces a *set* of operand-value tuples. Possible test values for `add` might be $V = \{(1, \texttt{rmode}(3), 5), (12, \texttt{imode}(-1024), 14)\}$, where each tuple contains the values for `rs1`, `reg_or_imm`, and `rd`.

Constructors with multiple branches usually have different encodings depending on which branch is chosen. A constructor's branches are represented by an ordered list of equations and pattern pairs, i.e., $(E_1, P_1) \ldots (E_m, P_m)$. Each E_j includes equations defined explicitly in a specification and equations derived by the toolkit's equation solver, e.g., to guarantee that solutions are integers. For an instruction with branches, we want to check (at least) one encoding of each branch (E_j, P_j). The SPARC `set` constructor, for example, has three branches, each of which should be checked. Checking branch j requires computing a tuple T_j that satisfies E_j, but *does not* satisfy $E_1 \ldots E_{j-1}$. The qualification is necessary because the toolkit chooses the first branch whose equations are satisfied, so if T_j satisfies some $E_i, i < j$, the encoding P_i, not P_j, will be checked. For example, checking the second branch of the `set` constructor requires not only that the `val` operand fit in 13 signed bits, but also that one of its least-significant 10 bits be nonzero, lest the first branch be chosen. Simply computing a tuple T_j that satisfies E_j is NP-complete; guaranteeing that T_j does not satisfy some other E_i is at least as hard. We give an NP-completeness proof by reduction from the zero-one integer programming problem [8] in the appendix.

We could search for satisfying test values by using one of the many exact or approximate algorithms for solving integer linear-programming (ILP) problems. Branch-and-bound algorithms are potentially exponential in the number of variables and constraints, but they are rarely exponential in practice [14, p 435]. Unfortunately, these algorithms require solutions to linear-programming problems over the reals at each branch step, and, therefore, require an implementation of non-trivial LP solvers. Pseudo-polynomial ILP algorithms are linear in the size of the coefficients of variables, but they are complex and difficult to implement. Application-specific algorithms exist for identifying data dependences across loops in parallelizing compilers [12, 15]. Their goals differ from ours: deciding an ILP problem, i.e., simply determining whether a solution exists, is sufficient for determining

	Total Constructors	Explicit Equations with Equalities	Inequalities
MIPS	103	43	0
SPARC	174	34	0
Pentium	586	3	7

Table 1: Equations in example specifications.

```
constructor_test_values(C) =
  V = {}
  for each branch (E_j, P_j) in C
     try {
        V' = branch_test_values(C, E_j)
        if every v ∈ V' satisfies some E_i, i < j then
           Warn that E_j is not checked
        else V = V ∪ V'
     } catch (Failed)
        Warn that E_j not checked
  return V
```

Figure 3: Selection of operand values for constructor C

whether optimizations can be applied to loops. In our testing problem, the ILP solution must be computed. Algorithms exist for computing ILP solutions to data-dependency problems, but they are highly application-specific. Another alternative is having the checker use an external ILP solver [7], but this requires that our users obtain the solver software, reducing the likelihood they will use the checker.

If most machine specifications included large sets of complex equations, we could justify the effort required to implement an ILP algorithm. In practice, however, most instructions have no explicit equations, and the equations derived by the solver are simple. Table 1 reports the number and type of equations in our example specifications. In our SPARC, MIPS, and Intel Pentium specifications, the largest equation set has three equations, and the equations refer to a total of three variables. The explicit equations are mostly equalities, for which it is easier to find satisfying values. For these reasons, we implement a simple algorithm that may, in principle, fail to find satisfying values for a branch, but in practice, finds satisfying operand values for every instruction in our MIPS, SPARC, and Intel Pentium specifications.

Figure 3 outlines our algorithm for computing a set of test values for constructor C. The real work is done by **branch_test_values**, which, given the equations for a single branch, computes a set of test values for that branch. If it is unable to compute a set of values, it raises the exception "Failed." As noted above, a set of values satisfying the constraints on a branch j might also satisfy the constraints on an earlier branch i, in which case using those values would exercise branch i and not branch j. Our search algorithm calls **branch_test_values** once to find values that satisfy E_j, but it cannot guarantee those values will not satisfy any of $E_1 \ldots E_{j-1}$. If values satisfying E_j also satisfy an earlier E_i, the toolkit will issue a warning message, but this has never occurred in practice.

Selecting Operand Values to Satisfy Equations

Figure 5 gives the pseudo-code for **branch_test_values**, which looks for an assignment to C's operands that satisfies E. Only integer, relocatable, and field operands occur in equations, so we find assignments to those operands first, then make recursive calls to **constructor_test_values** to find values for constructor-typed operands. One recursive call is made for each constructor in the type. For example, given a `reg_or_imm` operand, we call **constructor_test_values** on both the `rmode` and `imode` constructors. **branch_test_values** returns V, the set of test-value tuples for C, which includes all possible combinations of the values computed for each operand v_i.

The code uses the associative array V' to bind expressions to variables or slices of variables. The variables include operands as well as temporaries that are defined in E. We call an expression *computable* if it is a function of constants and variables with known values in V'. V' maps variables to computable expressions or to the special value "unknown." Initially, each variable v in V' is unknown, and $F[v]$, the set of v's values that are known not to satisfy E, is empty.

The search for an assignment is shown in the fragment ⟨**Select values for unknown variables**⟩, which chooses an assignment by deciding one binding at a time. Some bindings are "forced" by equations; others are chosen at random. Bindings are forced by equations that constrain variables to be equal to computable expressions. For example, if E contains the equation $v = v' + 4$, and if a binding for v' has already been determined, then this equation forces the binding of v.

After adding all forced bindings to V, the algorithm tries tentative bindings for variables not yet bound. The checker picks a binding $v = r$ at random, then re-runs the solver to check if $v = r$ is consistent with E. If so, equation $v = r$ forces a binding; moreover, the solver may have rewritten the equations to force other bindings as well. For example, if E contains the equation $v = v' + 4$, the solver will produce the forced binding $v' = r - 4$. If adding $v = r$ to E leads to a contradiction, the algorithm withdraws the equation, marks $v = r$ as forbidden (by adding r to $F[v]$) and tries a different random binding, repeating the search until LIMIT values have been forbidden. We use a LIMIT of 1024, but in our example specifications, fewer than a dozen constructors require LIMIT > 1, and no constructor requires LIMIT > 5. The process of binding a variable at random, refining equations, and finding forced bindings is repeated until all variables are bound to computable expressions. The checker then generates code to produce the integer value of each computable expression at run time.

The one value that cannot be determined by the checker is the program counter, because it represents the absolute address at which the instruction under test is located, so its value is not under the checker's control. The algorithm handles this problem by binding it to `pc`, the toolkit's abstract representation of the current location in the instruction stream. The toolkit provides the value of `pc` at run time, where it is used to help encode instructions like relative branches.

E	Equations
v	Variable
c	Constructor instance
$V'[v]$	Current test value for variable v
$V[v]$	Final test value for variable v
$F[v]$	Set of failed values for variable v
$W[v]$	Set of values in v's range

Figure 4: Variables used to select test values

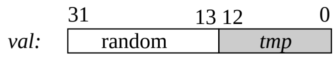
branch_test_values$(C,\ E)$=
  for each integer, field, or relocatable operand $v \in C$
    and each temporary variable $v \in E$ do {
    $V'[v]$:= unknown
    $F[v]$:= {}
  }
  $V'[pc]$:= `pc`
  ⟨**Select values for unknown variables**⟩
  for each constructor-typed operand $v \in C$ do
    $V'[v] := \bigcup_{c\ isa\ v}$ **constructor_test_values**(c)
  return V := sets_to_tuples($V'[v_1],\ V'[v_2],\ \ldots,\ V'[v_n]$)

⟨**Select values for unknown variables**⟩ =
  Get forced bindings from equations E
  while ($\exists\ V'[v]$ = unknown) {
    Select v with smallest range constraints in W
    repeat {
      Select value $r \in (W[v] - F[v])$
      Add $\{v = r\}$ to E and run solver
      if ($\{v = r\}$ is consistent with E) then
        Get forced bindings from revised equations E
      else {
        Remove $\{v = r\}$ from E
        Add r to $F[v]$
        Examine revised E and possibly select new v
      }
    } until (v is successfully bound or $|F[v]| >$ LIMIT)
    if ($|F[v]| >$ LIMIT) then raise Failed
  }

Figure 5: Selecting test values for branch (E, P)

We use a heuristic to select which variable should be bound on each iteration of the value-selection loop. To choose a variable, we examine W, the range constraints on field and integer operands that appear in patterns. We choose the unknown variable v that has the smallest range constraint. If $W[v]$ has fewer elements than LIMIT, and if an acceptable binding for v exists, we are guaranteed to find it.

Some equations constrain only some bits in a variable, which might lead to bindings of a range of bits. If ranges of bits within v are already bound when we try a binding $v = r$, we keep the values of the bound bits and use r to supply the unbound bits. For example, the second branch of the SPARC's `set` instruction is encodable only when `val` fits in 13 signed bits. The solver derives this condition through the equations $E = \{\texttt{val} = tmp, tmp\downarrow_{13} = \texttt{val}[0{:}12]\}$. The range sets of these values are $W[tmp] = \{-2^{12} \ldots 2^{12} - 1\}$ and $W[\texttt{val}] = \{0 \ldots 2^{32} - 1\}$. Here, the checker prefers tmp and selects a value for it first. This leads to a binding of $\texttt{val}[0{:}12] = tmp$. So the low-order 13 bits of `val` are determined by tmp, but its 18 high-order bits are random:



We could use more ambitious methods to search for bindings, for example, by using the properties of contradictory equations to choose a better value for v, but in practice, our simple technique works well for the equations that occur in machine specifications. If we find that new specifications have more complex constraints, simple improvements to our technique are possible. For example, many ILP algorithms preprocess equations using Banerjee's GCD test [3], which reduces the number of variables and constraints, and for equality constraints, can answer the ILP decision problem in polynomial time. Although we expect users to write equations that have integer solutions, the GCD test is an easy way to tell the user if no integer solution exists.

TESTING INSTRUCTIONS

After generating test values for every branch of every instruction, we produce a program that checks the encodings. The testing strategy in Figure 2, however, has several problems. First, it assumes that assembly and disassembly are exact inverses, whereas in practice, they are usually approximate inverses. For example, one assembly instruction may assemble into multiple machine instructions, and one machine instruction may have multiple representations in assembly. If we denote the assembly and binary forms by A and B, and we denote the assembly and disassembly transformations by $\alpha : A \to B$ and $\delta : B \to A$, then $\alpha \circ \delta$ may be the identity function on binaries, but $\delta \circ \alpha$ is seldom the identity function on assembly language, because assembly languages usually have more than one way to represent some instructions. Worse, the output of the disassembler δ may not be in A at all, i.e., it may not be valid assembly language. In that case, α and δ are neither right nor left inverses, and we can only compare in B. Comparing in B is sufficient to establish the presence of an inconsistency, but it is more useful to compare in A, because assembly language is designed for humans to read. Differences in A not only establish the presence of an error, but may show where in the specification the error lies and what the error is.

A final problem is that B is not a simple sequence of instructions but an object file, in which the instructions are surrounded by headers and other information, the format of which depends on the target machine and operating system. Writing code to emit object-file headers is tedious, but there is a better way.

The checker exploits the fact that every binary token in B has a representation in A that is independent of the toolkit specification and the instruction mnemonics. That representation is the pseudo-operations the assembler uses to place raw data into an instruction stream. For example, the Pentium instruction `"addb $127,%al"` is represented by two 8-bit tokens with values 4 and 127; the GNU assembler can emit those tokens with the pseudo-operations

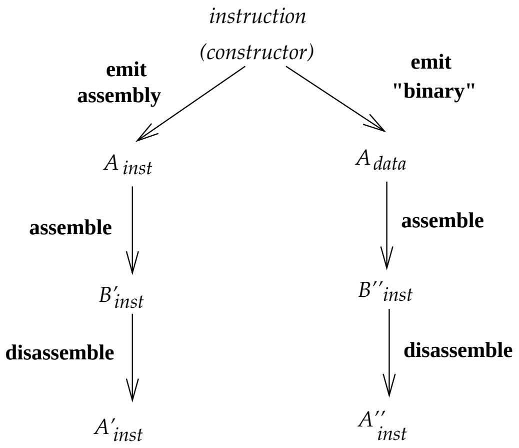


Figure 6: Mappings performed by checker

".byte 0x4; .byte 0x7f". We call these pseudo-ops A_{data}, and we provide the toolkit with "binary" emitters that emit pseudo-ops in A_{data}. The toolkit's assembly procedures emit assembly instructions using the native mnemonics, which we call A_{inst}. The checker compares the two versions by assembling both into B'_{inst} and B''_{inst} and then by disassembling the binary representations into A'_{inst} and A''_{inst}, as shown in Figure 6. The toolkit uses the assembler to produce the appropriate object file format and the disassembler to produce readable code. It does not matter if the disassembler maps into some language that is not a valid input to the assembler. The comparison in A is only read by the user and shell scripts, so any human-readable A is sufficient.

If the specification's assembly-to-binary mappings are correct, A'_{inst} and A''_{inst} should be identical. If they are not, the checker flags mismatched instructions, using a file-comparison tool like `diff`; because differences are printed in assembly language, the user can easily identify the source of the discrepancy in the original specification.

Figure 7 depicts the implementation of this strategy. The checker causes the toolkit to generate a single interface for the instruction set, with two implementations. The `Adata` implementation emits the "binary" form using pseudo-ops, and the `Ainst` implementation emits the native instruction mnemonics. The checker generates a C program that calls the procedures in each implementation.[4] Each procedure corresponds to a constructor, and the checker exercises each procedure on each test-value tuple in the V for that constructor. The assembly language emitted in this process must be preceded by short, target-dependent assembly headers, which make the output valid assembly language and disable instruction scheduling if necessary. These headers may be written by hand or copied from the output of a native compiler.

[4] The toolkit puts each implementation in an interface record, i.e., a structure containing function pointers, and the checker generates code that emits instructions by calling indirectly through a pointer to the interface record. That code can change implementations at run time by using a pointer either to `Adata` or to `Ainst`.

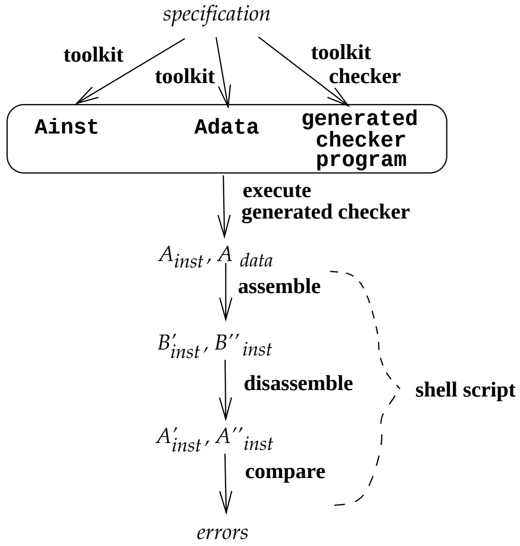


Figure 7: Steps to check specification.

The C program generated by the checker is linked with the encoding procedures in `Ainst` and `Adata`. When executed, it produces an assembly file that contains A_{data} followed by A_{inst}. A short shell script does the rest of the work. It applies the assembler to the file containing A_{inst} and A_{data}, applies the disassembler to B'_{inst} and B''_{inst}, and compares the resulting assembly files in A'_{inst} and A''_{inst}, and finally reports discrepancies to the user.

Some architectures, like the Pentium, are so popular that many different assemblers are available, each with its own syntax. Although the toolkit normally derives assembly-language syntax from the punctuation in constructor definitions, SLED supports specifications of alternate assembly-language syntax. A user can specify new syntax for any or all constructors without having to change the original specification. This modularity removes the temptation to clone the specifications of the binary representations, increasing reuse of the core specification. The user can choose the desired assembly syntax when the checker generates the assembly-language emitters.

RESULTS

We applied the checker to the Intel Pentium, MIPS, and SPARC specifications and found errors in all three. We found errors even after code generated from the MIPS and SPARC specifications had been used in one application for over a year. We were surprised at first, but then we realized that the application, a code generator, did not exercise all constructors in the specifications. Successful use of toolkit-generated code in an application fooled us into believing the specifications were correct; the checker showed us that a more comprehensive, independent testing procedure was necessary.

The checker found three types of errors: incorrect opcode values, incorrect operand order and type, and incorrect or missing equational constraints. The Pentium has large opcode tables, and in several cases, we mis-transcribed their contents while writing the specification. In several places, we reversed the order of constructors' operands. The effect of this error is that the toolkit produces a signature for an encoding procedure in which the operand order does not match the assembly-language syntax for the instruction, which can result in incorrect use of the encoding procedure by a user. We also incorrectly specified some unsigned operands as signed.

The checker also identified register-use conventions. Both the MIPS and SPARC require an even destination register for their double-word load and store instructions, because the destination registers are accessed as even-odd pairs. An attempt to load a double-word instruction from an odd destination register may cause an "illegal instruction" trap [23, p 91]. The MIPS manual does not specify the effect of a load to an odd register, but its assembler enforces the even-register programming convention. This convention can be expressed in SLED by adding the equation `rd = 2 * _` to the relevant constructor specifications. The checker identified similar register-use conventions for the double-word and quad-word operands to the MIPS and SPARC floating-point instructions. This was an unexpected benefit—documentation on conventions is often missing or hard to find, but with the checker, we were able to identify them quickly.

Some discrepancies were not real errors, but false positives. An aggressive assembler can introduce differences in the assembly and binary representations that are not caused by inconsistencies between the toolkit specification and the actual machine. For example, on the Pentium, the `ShortIndex` effective address `4[%eax*1]` is more compactly encoded by the `Disp32` effective address `4[%eax]`, and when the longer `ShortIndex` form is expressed in A_{inst}, the assembler rewrites it into the shorter `Disp32` form before encoding it. The toolkit, by contrast, does exactly as it is told and encodes the longer version into A_{data}. The result is a "false" mismatch after both the A_{inst} and A_{data} versions are assembled into B. The checker helps identify assembler idioms like these; if the user wants his specification to support such idioms, he can rewrite the idiomatic constructors with branches that specify the various encodings.

EVALUATION

Our goals are to make applications that process machine code easier to write, easier to debug, and easier to retarget to new architectures. The toolkit has met these goals in several ways. The toolkit allows programmers to write at an assembly-language level of abstraction, which eliminates the tedium of manipulating machine code directly and allows programmers to focus on their applications. SLED supports retargeting by simplifying specification of a new architecture and by providing a formal description that corresponds to the informal description in an architecture manual. The toolkit checks for implausible and impossible specifications, which increases the utility of specifications and shifts much of the effort of debugging from run time to specification-compile time. Finally, code generated by the toolkit enforces the constraints required to encode and decode instructions correctly.

The checker is a vital part of the toolkit, because it helps guarantee that the encoding and decoding code derived from a specification are acceptable replacements for a vendor's assembler and disassembler. Although the problem of guaranteeing that all instruction encodings in a specification are checked is NP-complete, most equational constraints in specifications are simple. Our heuristic algorithm was able to check all instructions in the MIPS, SPARC, and Intel Pentium. We simplified the implementation effort by using the toolkit's equation solver to help select test values and by using an independent assembler as a validator. Most importantly, the checker identified lingering bugs in "good" specifications, which reinforced the importance of testing specifications systematically instead of declaring victory after building a working application.

AVAILABILITY Version 0.5 of the toolkit is available by anonymous `ftp` from `ftp.cs.princeton.edu` in directory `pub/toolkit`. The toolkit's home page on the World-Wide Web is `http://www.cs.princeton.edu/software/toolkit`.

ACKNOWLEDGEMENTS We thank the anonymous referees for their suggestions, and especially for telling us about the buggy microcode in the C-30 IMP. The second author was supported in part by NSF Grant ASC-9612756 and DARPA Contract MDA904-97-C-0247.

REFERENCES

[1] J. Auslander, M. Phillipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–158, Philadelphia, PA, June 1996.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(3):1319–1360, July 1994.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[4] O. C. Braun. Retargetability issues in worst-case timing analysis of embedded systems. Bachelor's thesis, Dept of Computer Science, Princeton University, May 1996.

[5] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[6] M. F. Fernández. Simple and effective link-time optimization of Modula-3 programs. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 30(6):103–115, June 1995.

[7] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL : A Modeling Language for Mathematical Programming.* Boyd & Fraser Pub. Co., 1994.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman, San Francisco, CA, 1979.

[9] L. George, F. Guillame, and J. H. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *5th International Conference on Compiler Construction*, pages 83–97, Apr. 1994.

[10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, San Francisco, CA, Jan. 1992.

[11] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, PA, June 1996.

[12] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, Toronto, Canada, June 1991.

[13] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[14] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[15] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[16] N. Ramsey. Relocating machine instructions by currying. *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):226–236, May 1996.

[17] N. Ramsey. A simple solver for linear equations containing nonlinear operators. *Software—Practice & Experience*, 26(4):467–487, Apr. 1996.

[18] N. Ramsey and M. F. Fernández. New Jersey Machine-Code Toolkit architecture specifications. Technical Report TR-470-94, Department of Computer Science, Princeton University, Oct. 1994.

[19] N. Ramsey and M. F. Fernández. New Jersey Machine-Code Toolkit reference manual. Technical Report TR-471-94, Department of Computer Science, Princeton University, Oct. 1994.

[20] N. Ramsey and M. F. Fernández. The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA, Jan. 1995.

[21] N. Ramsey and M. F. Fernández. Specifying representations of machine instructions. To appear in *ACM Transactions on Programming Languages and Systems.*, 1997.

[22] N. Ramsey and D. R. Hanson. A retargetable debugger. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 27(7):22–31, July 1992.

[23] SPARC International, Englewood Cliffs, NJ. *The SPARC Architecture Manual, Version 8*, 1992.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Dec. 1993.

Proof of TEST $\in$ NP-Complete

We show that the testing problem (TEST) is NP-Complete by reduction from Zero-One Integer Linear Programming (Zero-One ILP). An instance of TEST is a constructor D applied to the tuple of operands, V. TEST determines if there exists a value assignment for V that satisfies one branch of D's conditions and therefore guarantees one encoding of D will be tested.

TEST is in NP, because a non-deterministic algorithm can guess a value assignment V' for V, select a branch to test, and check in polynomial time that V' satisfies that branch. Recall that the values of field operands to an instruction are bounded by the width of the field and that integer operands are bounded by the word size of the target machine.

We prove TEST is NP-complete by giving a polynomial time reduction from Zero-One ILP [8] to TEST. An instance of Zero-One ILP is $\bar{x} = (x_1, ..., x_m)$ and $\bar{c} = (c_1, ..., c_m)$, both m-tuples of integers in $\{0, 1\}$, b and B, both integers in $\{0, 1\}$, and $X = \{(\bar{x_1}, b_1), ..., (\bar{x_n}, b_n)\}$, a set of pairs $(\bar{x}, b)$. Zero-One ILP determines if there exists an m-tuple $\bar{y}$ of integers in $\{0, 1\}$, such that $\bar{x} \cdot \bar{y} \leq b$ for all $(\bar{x}, b) \in X$ and that $\bar{c} \cdot \bar{y} \geq B$.

We represent an instance of Zero-One ILP by the constructor `ZILP` and each element y_i in $\bar{y}$ by a field operand v_i in V. For each pair $(\bar{x_k}, b_k) \in X$ we add the constraint $x_{k1} \times v_1 + x_{k2} \times v_2 + ... x_{km} \times v_m <= b_k$ to `ZILP`'s equations; this constraint implements $\bar{x} \cdot \bar{y} \leq b$. We also add the constraint $c_1 \times v_1 + c_2 \times v_2 + ... c_m \times v_m >= B$. For example, `ZILP` is defined as:

```
fields of v(n) v1 0:0 v2 1:1 ... vn (n-1):(n-1)
fields of t(1) testable 0:0
constructors
  ZILP (v1, v2, ..., vm)
     { x11*v1 + x12*v2 + ... x1n*vm <= b1,
       ...
       xn1*v1 + xn2*v2 + ... xnm*vm <= bm,
       c1*v1  + c2*v2  + ... cm*vm  >= B }
  is testable = 1
```

where `xkj` is the j-th element of the tuple x_k.

We show that `ZILP` is testable if and only if there exists $\bar{y}$ as defined by the Zero-One ILP. Suppose there exists a V such that `ZILP` is testable. Then, by our definition of testable, V satisfies `ZILP`'s single branch, and `ZILP`'s encoding is `testable = 1`. Since there is a one-to-one correspondence between V and $\bar{y}$ and between `ZILP`'s equations and the constraints in the Zero-One ILP, the solution to the Zero-One ILP is $\bar{y} = V$.

Suppose that there exists a $\bar{y}$ that satisfies the above constraints. Again, by the one-to-one correspondence between $\bar{y}$ and V and between the constraints in the Zero-One ILP and `ZILP`'s equations, `ZILP`'s equations are satisfied by V and its single branch is testable.