# Specifying Representations of Machine Instructions

NORMAN RAMSEY
University of Virginia
and
MARY F. FERNÁNDEZ
AT&T Labs

We present SLED, a Specification Language for Encoding and Decoding, which describes abstract, binary, and assembly-language representations of machine instructions. Guided by a SLED specification, the New Jersey Machine-Code Toolkit generates bit-manipulating code for use in applications that process machine code. Programmers can write such applications at an assembly-language level of abstraction, and the toolkit enables the applications to recognize and emit the binary representations used by the hardware. SLED is suitable for describing both CISC and RISC machines; we have specified representations of MIPS R3000, SPARC, Alpha, and Intel Pentium instructions, and toolkit users have written specifications for the Power PC and Motorola 68000. The article includes representative excerpts from our SPARC and Pentium specifications. SLED uses four elements; *fields* and *tokens* describe parts of instructions; *patterns* describe binary representations of instructions or groups of instructions; and *constructors* map between the abstract and binary levels. By combining the elements in different ways, SLED supports machine-independent implementations of machine-level concepts like conditional assembly, span-dependent instructions, relocatable addresses, object code, sections, and relocation. SLED specifications can be checked automatically for consistency with existing assemblers. The implementation of the toolkit is largely determined by our representations of patterns and constructors. We use a normal form that facilitates construction of encoders and decoders. The article describes the normal form and its use. The toolkit has been used to help build several applications. We have built a retargetable debugger and a retargetable, optimizing linker. Colleagues have built a dynamic code generator, a decompiler, and an execution-time analyzer. The toolkit generates efficient code; for example, the linker emits binary up to 15% faster than it emits assembly language, making it 1.7–2 times faster to produce an `a.out` directly than by using the assembler.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*systems specification methodology*; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*; D.3.4 [**Programming Languages**]: Processors—*translator writing systems and compiler generators*

General Terms: Languages

Additional Key Words and Phrases: Compiler generation, decoding, encoding, machine code, machine description, object code, relocation

This work has been funded by a Fannie and John Hertz Fellowship, an AT&T Ph.D. Fellowship, an IBM Graduate Research Fellowship, and by Bellcore.
Authors' addresses: N. Ramsey, Department of Computer Science, University of Virginia, Charlottesville, VA 22903; email nr@cs.virginia.edu; M. F. Fernández, AT&T Labs — Research, 180 Park Avenue, Florham Park, NJ 07932-0971; email mff@research.att.com.

## 1. INTRODUCTION

This article describes SLED—Specification Language for Encoding and Decoding—and its implementation in The New Jersey Machine-Code Toolkit. SLED specfications define mappings between symbolic, assembly-language, and binary representations of machine instructions. We have specified representations of MIPS R3000, SPARC, Alpha, and Intel Pentium instructions; toolkit users have written specifications for the Power PC and Motorola 68000. The specifications are simple, which makes it practical to use the toolkit to write applications for multiple architectures. The toolkit uses SLED specifications to help programmers write applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, and debuggers. The toolkit lets programmers encode and decode machine instructions symbolically. Guided by a SLED specification, it transforms symbolic manipulations into bit manipulations.

Traditional applications that process machine code include compilers, assemblers, linkers, and debuggers. Newer applications include profiling and tracing tools [Ball and Larus 1992; Cmelik and Keppel 1994], testing tools [Hastings and Joyce 1992], protection enforcers [Wahbe et al. 1993], run-time code generators [George et al. 1994], and link-time optimizers [Fernández 1995; Srivastava and Wall 1993]. There are even some frameworks for creating applications that manipulate executable files, although none that work on more than one machine [Johnson 1990; Larus and Schnarr 1995; Srivastava and Eustace 1994]. Graham et al. [1995] describe auxiliary information needed to facilitate machine-code manipulations; they report support for the MIPS and SPARC architectures.

A few applications avoid machine code by using assembly language, e.g., many Unix compilers emit assembly language, not object code. It is not always practical, however, to use an assembler, e.g., when generating code at run time or adding instrumentation after code generation. Some machine-code applications can be duplicated by source-code transformation; such applications usually work on many machines, but they cannot be used as often as applications that work on object code, because source code is not always available. Our toolkit makes it easier to build applications and frameworks that work with object code and therefore can be used on any executable file.

Applications that cannot use an assembler currently implement encoding and decoding by hand. Different ad hoc techniques are used for different architectures. The task is not intellectually demanding, but it is error prone; bit-manipulating code usually harbors lingering bugs. Our toolkit automates encoding and decoding, providing a single, reliable technique that can be used on a variety of architectures. It is intended less to support traditional compilation than to support nontraditional operations like rewriting executable files or run-time code generation.

Applications use the toolkit for encoding, decoding, or both. For example, assemblers encode; disassemblers decode; and some profilers do both. All applications work with *streams* of instructions. Decoding applications use *matching statements* to read instructions from a stream and identify them. A matching statement is like a case statement, except its alternatives are labeled with patterns that match instructions or sequences of instructions. Encoding applications call C procedures generated by the toolkit. These procedures encode instructions and emit them into a stream, e.g., the SPARC call `fnegs(r2, r7)` emits the word `0x8fa000a2`. Streams can take many forms; for example, a debugger can treat the text segment of a target process as an instruction stream. The toolkit's library provides a representation of streams that should be convenient for many encoding applications.

The toolkit has four parts. The *translator* takes a program with embedded matching statements and translates these statements into ordinary code. It handles programs written in C or Modula-3 [Nelson 1991]. The *generator* generates encoding and relocation procedures in C. These procedures call code in the *library*. The library implements both instruction streams and relocatable addresses, which refer to locations within the streams. The *checker* checks specifications for consistency with existing assemblers. The translator, generator, and checker need an instruction specification; encoding procedures and checking code are generated from the specification, and matching statements can match the instructions or parts thereof defined in the specification. The library is machine independent.

The SLED specification language is simple, and it is designed so that specifications can resemble instruction descriptions found in architecture manuals. SLED uses a single, bidirectional construct to describe both encoding and decoding, so their consistency is guaranteed. The toolkit checks specifications for unused constructs, underspecified instructions, and internal inconsistencies. An instruction encoding can be specified with modest effort; our Alpha, MIPS, SPARC, and Pentium specifications are 118, 127, 193, and 460 lines. The SLED specification language is the primary subject of this article.

Simplicity in specification is more than a personal preference. Simple specifications are more likely to be correct, and correct specifications are more valuable if they can be used in a variety of applications. To make the toolkit simple and general, we avoid describing the semantics of instructions, because too often semantic information is both hard to get right and of use only to a single application. Instead, SLED focuses describing an abstract representation of instructions and on automating the translation to and from that abstract representation.

We have personal experience with two applications that use the toolkit. `mld`, a retargetable, optimizing linker [Fernández 1995], uses the toolkit to encode instructions and emit executable files. `ldb` [Ramsey 1992; Ramsey and Hanson 1992], a retargetable debugger, uses the toolkit to decode instructions and to implement breakpoints. Others have used the toolkit to help develop a run-time code generator, a decompiler, an execution-time analyzer [Braun 1996], and an optimizing compiler for object-oriented languages [Dean et al. 1996].

Using the toolkit reduces retargeting effort and makes code more reliable. For example, `ldb`'s disassembler for the MIPS requires less than 100 lines of code, and `mld` has replaced 450 lines of hand-written MIPS code with generated encoding and relocation procedures. By hiding shift and mask operations, by replacing case statements with matching statements, and by checking specifications for consistency, the toolkit reduces the possibility of error. The toolkit can speed up applications that would otherwise have to generate assembly language instead of binary code. For example, `mld` creates executable files 1.7 to 2 times faster when using toolkit-generated encoding procedures than when using assembly language and calling a native assembler. To realize such speedups without the toolkit, `mld` would need hand-written encoding and relocation procedures for each target architecture.

The primary contribution of our work is the SLED specification language, which is expressive enough to write clear, concise, and reusable specifications of instruction representations for a variety of widely used architectures. Our processor for these specifications derives code for both encoding and decoding problems, eliminating a significant source of retargeting effort. Our model of machine instructions makes several machine-level concepts general enough to be specified or implemented in a machine-independent way. These concepts include conditional assembly, span-dependent instructions, relocatable addresses, object code, sections, and relocation.

Most of this article is devoted to SLED. We begin with an extended example: a specification for a representative subset of the SPARC instruction set. This example shows how a typical specification is structured and how SLED is used idiomatically. We then cover the details of syntax, semantics, and implementation, followed by smaller examples from our Pentium specification, which show CISC addressing modes and variable-sized operands. We explain how applications use the code generated by the toolkit, and we conclude with a discussion of related work and an evaluation of the toolkit and its specification language.

## 2. SPECIFYING INSTRUCTION REPRESENTATIONS

To illustrate SLED, we specify a subset of the SPARC instruction set. The illustration is drawn from our complete, annotated specification of the SPARC [Ramsey and Fernández 1994a]. It includes the SPARC's integer instructions, but it omits floating-point instructions, several types of load and store, and many synthetic instructions. Before beginning the illustration, we explain the elements of the specification language and our strategy for using the language to describe a machine.

Because machine instructions do not always fit in a machine word, the toolkit works with streams of instructions, not individual instructions. An instruction stream is like a byte stream, except that the units may be "tokens" of any size, not just 8-bit bytes. An instruction is a sequence of one or more tokens, so "token stream" might be a more precise term. Tokens may come in any number of *classes*, which help distinguish different parts of complex instructions. For example, a Pentium instruction might include several 8-bit prefixes, an 8-bit opcode, 8-bit format bytes, and a 16-bit immediate operand. Most likely, the prefixes and opcode would be tokens from the same class, but the format bytes and operand would be from different classes.

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. Tokens of a single class may be partitioned in more than one way. *Patterns* constrain the values of fields; they may constrain fields in a single token or in a sequence of tokens. Patterns describe binary representations of instructions, groups of instructions, or parts of instructions. For example, simple patterns can be used to specify opcodes, and more complex patterns can be used to specify addressing modes or to specify a group of three-operand arithmetic instructions.

*Constructors* connect abstract, binary, and assembly-language representations of instructions. At an abstract 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 that gives the instruction's binary representation, which is typically a sequence of tokens. Each constructor is also associated with a function that produces a string, which is the instruction's assembly-language representation. Specification writers use constructors to define an abstract 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.

Machine designers might expect binary representations to be untyped. We have found it useful to associate type information with binary representations or with fragments of binary representations, for the same reason that programming languages do so—to help detect and prevent errors. The classes of tokens are like types. We also require that each constructor have a type. We provide a predefined, anonymous type for constructors that produce whole instructions, and specification writers may introduce more constructor types. We typically use such types to describe effective addresses or structured operands. When used in this way, the constructor type corresponds to the "operand class" of Cattell [1980], and each constructor of the type corresponds to one "access mode." The toolkit maps constructor types onto types in the code it generates, which helps find errors in application code as well as in specifications.

To describe a machine, we begin by specifying tokens and fields, which are the basic components of instructions. Next come patterns that specify opcodes and groups of related opcodes, then constructors that specify structured operands, like effective addresses. Having specified opcodes and operands, we define constructors that specify instructions. When possible, we specify many constructors concisely by using "opcode patterns," which group related instructions.

Many architecture manuals use the term "synthetic" to describe instructions that are supported by an assembler, but not directly by the hardware. The assembler synthesizes such instructions by using special cases or combinations of other instructions. SLED specifications can include synthetic instructions, for which binary representations are given by applying previously defined constructors. We typically specify synthetic instructions in a separate file, since they are useful only in some applications.

The rest of this section gives excerpts from our specification of the SPARC. We have engineered SLED's syntax to foster resemblances between specifications and architecture manuals, and we refer to relevant pages of the SPARC manual [SPARC International 1992] by page number. When concatenated, the excerpts form a complete SLED specification for a subset of the SPARC architecture. The specification is included in the toolkit's source distribution.

We use bit numbers to specify the positions of fields within tokens. Since different manuals use different conventions, the toolkit supports both little-endian and big-endian bit numberings. The SPARC manual uses the little-endian numbering.

```
bit 0 is least significant
```

Architecture manuals usually have informal field specifications. For example, the fields for some SPARC load instructions are [SPARC International 1992, p. 90]:

| op | rd | op3 | rs1 | i | simm13 |
|---|---|---|---|---|---|
| 31 30 | 29 25 | 24 19 | 18 14 | 13 | 12 0 |

`fields` declarations give the locations and sizes of fields. The declaration below defines the fields used in all SPARC instructions. The first line defines the fields in the picture above. The remaining lines define all the other fields used in the SPARC manual, even those used only in floating-point instructions, which are otherwise omitted from this article.

```
fields of itoken (32)
  op 30:31 rd 25:29 op3 19:24 rs1 14:18  i 13:13 simm13 0:12
  disp30 0:29 op2 22:24 imm22 0:21 a 29:29 cond 25:28 disp22 0:21
  asi 5:12 rs2 0:4 opf 5:13 fd 25:29 cd 25:29 fs1 14:18 fs2 0:4
```

We often want to give auxiliary information about some fields, which we do with `fieldinfo` directives. This directive gives mnemonic names to the two possible values of the `a` field.

```
fieldinfo a is [ names [ "" ",a" ] ]
```

`a` is the "annul" bit, and the toolkit uses its names below to help derive the names of branch constructors.

Architecture manuals often define opcodes in tables. The SPARC manual uses a hierarchy of tables; we show specifications for several. Tables F-1 and F-2 [SPARC International 1992, p. 227] are specified by

```
patterns
  [ TABLE_F2 call TABLE_F3 TABLE_F4 ]    is op = {0 to 3}
  [ 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 `Bicc` is bound to `op = 0 & op2 = 2`. Bindings to the wildcard "_" are ignored. The second line of the excerpt corresponds to Table F-1, but the identifier `TABLE_F1` does not appear, because there are no references to Table F-1 from other tables.

Table F-3 [SPARC International 1992, 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 toolkit can handle opcode tables in row-major or column-major form. 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 the pattern `op = 2 & op3 = 16`, effectively using a column-major numbering.

Table F-4 [SPARC International 1992, p. 229] defines the load and store opcodes; it is specified by

```
[ ld     lda     ldf   ldc
  ldub   lduba   ldfsr ldcsr
  lduh   lduha   _     _
  ldd    ldda    lddf  lddc
  st     sta     stf   stc
  stb    stba    stfsr stcsr
  sth    stha    stdfq stdcq
  std    stda    stdf  stdc
  _      _       _     _
  ldsb   ldsba   _     _
  ldsh   ldsha   _     _
  _      _       _     _
  _      _       _     _
  ldstub ldstuba _     _
  _      _       _     _
  swap   swapa   _     _  ]
is
  TABLE_F4 & op3 = {0 to 63 columns 4}
```

Most operands to instructions are fields or integers, but some operands, like effective addresses, have more structure. We use typed constructors to define such operands. The address operands [SPARC International 1992, p. 84] have four possible formats:

```
constructors
  dispA     rs1 + simm13! : Address  is  i = 1 & rs1 & simm13
  absoluteA simm13!       : Address  is  i = 1 & rs1 = 0 & simm13
  indexA    rs1 + rs2     : Address  is  i = 0 & rs1 & rs2
  indirectA rs1           : Address  is  i = 0 & rs2 = 0 & rs1
```

Each line specifies a constructor by giving its opcode, operands, type, and pattern. Usually, as here, the opcode is simply the constructor's name. The plus signs among the operands indicate the preferred rendering of these constructors in assembly language. The operand specification `simm13!` indicates a signed integer operand destined for field `simm13`. Each of these constructors has type `Address`, which is effectively a disjoint union type containing an element for each constructor. We use the `Address` type below to specify operands of constructors for load and store instructions. When a field name is used as a pattern, as is `rs1` on the right-hand side of the `dispA` constructor, it is an abbreviation for the more verbose pattern `rs1 = rs1`, which forces the field `rs1` to be equal to the operand named `rs1`. This abbreviation appears frequently because operands are often placed directly in fields.

We also use typed constructors to specify "register or immediate" operands:

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

Architecture manuals often group definitions of related instructions, like the load-integer instructions in the SPARC manual [SPARC International 1992, p. 90]. We use disjunctions of patterns to represent such groupings, which can make specifications more concise. The specification

```
patterns loadg is ldsb | ldsh | ldub | lduh | ld | ldstub | swap
constructors
  loadg  [Address], rd
```

defines a group of untyped constructors, one for each general-purpose load instruction. The specification demonstrates two features of SLED: opcode expansion and implicit patterns. When the pattern `loadg` is given as the opcode in a constructor specification, it is expanded into individual disjuncts, and the construct is equivalent to repeated specifications of `ldsb`, `ldsh`, etc. Omitting the right-hand side tells the toolkit to compute a pattern by conjoining the opcode and all the operands. This idiom is ubiquitous in specifications of RISC machines. Finally, the square brackets and comma indicate assembly-language syntax.

These examples show how different elements of the specification interact. The constructor type `Address` is an abstraction representing "addressing mode." The four constructors of that type specify the different operands of addressing modes as well as their representations. The type `Address` is used in the `loadg` specification, so the load constructors take a first operand that represents an addressing mode. That operand must be the result of applying one of the four constructors of type `Address` defined above. For example, to load register `%l0` from a location on the stack, a compiler might make the call `loadg(dispA(r_fp, -12), r_l0)`. This example assumes that `r_fp` and `r_l0` are suitably defined constants.

We use the same techniques to specify the logical, shift, and arithmetic instructions, which take two register operands and one operand of type `reg_or_imm`. The last line specifies 38 constructors at once:

```
patterns
  logical is and  | andcc | andn | andncc
           | or   | orcc  | orn  | orncc
           | xor  | xorcc | xnor | xnorcc
  shift   is sll  | srl   | sra
  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
  alu is  arith | logical | shift
constructors
  alu rs1, reg_or_imm, rd
```

Using `reg_or_imm` as an operand means that the second operand to any of these constructors must have been produced by applying either the `imode` constructor or the `rmode` constructor defined above.

The first column of Table F-7 [SPARC International 1992, p. 231] defines branch opcodes:

```
patterns
  branch is any of
    [ bn be ble bl bleu bcs bneg bvs ba bne bg bge bgu bgeu bpos bvc ],
    which is Bicc & cond = {0 to 15}
```

This compound binding is a notational abbreviation that relieves us from writing the names in square brackets ("`bn be`...") twice. It both defines these names and makes `branch` stand for the pattern matching any of them.

To specify the branch *instructions*, we need two more features of SLED: relocatable operands and sets of equations. Designating an operand as *relocatable* means its value may be unknown at encoding time:

```
relocatable addr
```

If an application tries to encode an instruction with such an operand, and if the operand's value is unknown, the encoding procedure emits a *placeholder* for the instruction, together with a *relocation closure* that can be used to overwrite the placeholder when the missing value becomes known [Ramsey 1996a]. The most common example of such an instruction is a branch to an unknown label.

For convenience, we choose an invalid instruction as a placeholder. Because the execution of an invalid instruction causes a fault, it is easy to detect application bugs that cause placeholders to be executed:

```
placeholder for itoken is unimp & imm22 = 0xbad
```

Although the target address is an operand to a branch, it is not found in any field of the instruction; instead, it is computed by adding a displacement to the program counter. The equation in curly braces shows the relationship, which is taken from SPARC International [1992, pp. 119–120]:

```
constructors
  branch^a addr { addr = 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 `addr` and the program counter. The toolkit expands the 16 alternatives for `branch` and the two alternatives for `a`, so this line specifies 32 constructors.

We specify synthetic instructions by applying the constructors that correspond to the instructions from which they are synthesized. Here are definitions of `bset` (bit set) and `dec` (decrement) [SPARC International 1992, p. 86]:

```
constructors
  bset reg_or_imm, rd  is   or(rd, reg_or_imm, rd)
  dec  val!, rd        is  sub(rd, imode(val), rd)
```

The patterns on the right-hand sides are notated as constructor applications.

Some synthetic instructions may stand for more than one instruction sequence, depending on the values of operands. We specify such instructions by putting alternative *branches* on the right-hand side of a constructor specification. Each branch may have its own set of equations. The toolkit encodes the first possible branch whose equations have a solution and whose operand values fit in the fields to which they are bound. For example, the synthetic instruction `set` [SPARC International 1992, p. 84] expands to a single instruction when possible, but requires two in the general case:

```
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.

## 3. SLED SYNTAX AND SEMANTICS

Now that we have illustrated SLED with an extended example, we present its syntax and semantics in detail. We also describe the toolkit's internal representation in enough detail so that our techniques could be used in other systems.

SLED solves not only the intellectual problem of describing instruction representations, but also several practical problems in the generation of encoding and decoding applications. Throughout this section, we associate language constructs with problems that they solve, and we identify constructs that are motivated by the special needs of encoding, decoding, or other applications.

To describe syntax, we use an EBNF grammar with standard metasymbols for

$$\big\{\text{sequences}\big\},\ \big[\text{optional constructs}\big],\ \text{and}\ \big(\text{alternative}\ \big|\ \text{choices}\big).$$

We use large metasymbols to help distinguish them from literals. Terminal symbols given literally appear in `typewriter` font. Other terminal symbols and all nonterminals appear in *italic* font. Excerpts from the grammar always begin with the name of a nonterminal followed by the ⇒ ("produces") symbol.

*Specification* is the grammatical start symbol for SLED specifications. Within a specification, definitions must appear before uses, but otherwise the parts of a specification may appear in any order; so a specification is a list of *spec*:

$$\textit{specification} \Rightarrow \big\{\textit{spec}\big\}$$

### 3.1 Tokens and fields

The toolkit supports both little-endian and big-endian bit numberings.

$$\textit{spec} \Rightarrow \texttt{bit 0 is} \ \big(\texttt{most}\ \big|\ \texttt{least}\big)\ \texttt{significant}$$

The default numbering makes bit 0 the least-significant bit.

`fields` declarations specify how to divide tokens into fields. One `fields` declaration is given for each *class* of tokens; only fields named in the declaration can be extracted from tokens of that class. Each field appears in tokens of exactly one class. The `fields` declaration binds field names to bit ranges and specifies the number of bits in tokens of its class. The toolkit generates the shifts and masks needed to manipulate the value of a field in a token. The **fields** syntax is as follows:

$$spec \Rightarrow \texttt{fields of}\ \textit{class-name}\ \texttt{(}\ \textit{width}\ \texttt{)}\ \{\textit{field-name low-bit}{:}\textit{high-bit}\}$$

Field values are always unsigned; storing signed values in fields requires the explicit sign-extension operator, a postfix exclamation point. For example, this operator is applied to the displacement field `disp22` in the definition of the SPARC branch constructors. We make all field values unsigned because implicit sign extension can be confusing—people reading specifications should not have to remember which fields are signed and which are unsigned. Explicit sign extension also supports the use of the same field in different contexts with or without sign extension.

Fields solve the problem of specifying binary representations at the lowest level. They offer several advantages over bit strings, a more usual alternative. To make a token from bit strings, the strings must be concatenated in the right order; the order of fields is implicit in their declarations. One cannot assign the wrong number of bits to a field, and the toolkit detects cases in which fields overlap or leave gaps.

When instructions vary in size, more than one class of tokens may be needed. On the Intel Pentium, instructions are composed of 8-, 16- and 32-bit tokens, which must be given different classes because they are of different sizes. It can even be useful to put tokens of the same size in different classes. For example, the Pentium uses a "ModR/M" byte to specify addressing modes and an "SIB" byte to identify index registers [Intel Corporation 1993, p. 26-3]:

| | | | |
|---|---|---|---|
| ModR/M | mod | reg/opcode | r/m |
| | 7 6 | 5 3 | 2 0 |
| SIB | ss | index | base |
| | 7 6 | 5 3 | 2 0 |

The **fields** declarations for these bytes are

```
fields of ModRM (8)  mod 6:7 reg_opcode 3:5 r_m  0:2
fields of SIB   (8)  ss  6:7 index      3:5 base 0:2
```

Dividing tokens into classes helps detect errors in specifications. For example, putting the ModR/M and SIB tokens in different classes ensures that a user cannot mistakenly match both a `mod` field and an `index` field in the same byte.

One could also divide SPARC tokens into classes, e.g., by using a different class for each instruction format. One would have to define several replicas of fields that, like `op`, are common to multiple formats, because each field belongs to exactly one class. We judge that the extra effort would not pay off; the toolkit checks that the fields used in instructions partition the instructions' tokens, and this check seems adequate to detect errors on machines like the SPARC.

SLED specifications can include information about the names of field values and about the way fields are expected to be used in an application. The syntax used is as follows:

$$spec \Rightarrow \texttt{fieldinfo}\ \{\textit{field-specifier}\ \texttt{is [}\ \{\textit{field-item}\}\ \texttt{]}\}$$

$$\textit{field-specifier} \Rightarrow \textit{field-name}\ |\ \texttt{[}\ \{\textit{field-name}\}\ \texttt{]}$$

*field-item* ⇒ `sparse [` *binding* { `,` *binding* } `]`
  | `names [` { *Ident* | *String* } `]`
  | `checked` | `unchecked` | `guaranteed`

*binding* ⇒ ( *Ident* | *String* ) `=` *integer*

`sparse` and `names` specify names of fields. `names` is used when all values have names; `sparse` is used otherwise. Naming field values solves no single problem; the names are used in a variety of ways. The most unusual use may be SLED's use of field names in constructor specifications; when fields are used to specify constructor opcodes, the names of the values become part of the names of constructors. For example, our SPARC specification uses the names `""` and `",a"` for the values 0 and 1 of the `a` field, and these names become part of the names of branch constructors. The toolkit also uses the names when generating encoding procedures that emit assembly language and when generating disassemblers. Finally, the toolkit can generate tables of field names so applications can print names of field values.

The other information about fields helps solve the problem of generating efficient encoders. The toolkit normally checks field values at encoding time to be sure they can be represented in the number of bits available. These safety checks are needed only when field values are supplied by an application; no safety checks are generated when the toolkit can infer that values are representable. The checks can be fine-tuned using the `checked`, `unchecked`, and `guaranteed` attributes of fields. Application writers unwilling to pay for a compare and branch can designate fields as unchecked, in which case encoding procedures do not check their values but simply mask out high bits so tokens are not corrupted by bogus values. Those unwilling to pay even the cost of masking can designate fields as guaranteed, in which case their values are used without checking or masking; the application guarantees that the value fits. For example, code generators typically guarantee fields denoting registers, since the register allocator can easily ensure that register numbers fall in the proper range. Such a guarantee could be added to our SPARC example by writing

```
fieldinfo [ rs1 rs2 rd fs1 fs2 fd cd ] is [ guaranteed ]
```

Fields are checked by default.

### 3.2 Patterns

Patterns constrain both the division of streams into tokens and the values of the fields in those tokens. When instructions are decoded, patterns in matching statements identify interesting inputs; for example, a pattern can be defined that matches any branch instruction. When instructions are encoded, patterns in the machine specification specify what tokens are written into the stream.

Patterns are composed from *constraints* on fields. A constraint fixes the values a field may have. Constraints come in two forms: *range constraints* and *field bindings*. Range constraints are used when the values permitted in a field are known statically. Range constraints are represented internally in the form $lo \leq \mathtt{f} < hi$, forcing the value of the field to fall in a range. The external syntax is more restrictive; it requires that the field name be to the left of a single relational operator. The general form can be obtained by conjoining two constraints on the same field. The restricted syntax presents no burden in practice, because almost all range constraints use a range that contains one value, and we write them with an equals sign, e.g., `op = 1`.

Field bindings are used when the value of a field is not known until encoding time. A field binding forces a field to be equal to a value computed dynamically, and the dynamic computation is represented as an expression containing free variables. Field bindings are also written with equals signs.

Patterns are composed by conjunction (`&`), concatenation (`;`), and disjunction (`|`). They can also be computed by applying constructors. The syntax for patterns is as follows:

| | | |
|---|---|---|
| *pattern* ⇒ | *name* | *name* of pattern, field, or constructor type |
| | \| *field-name rel-op expr* | Constrains or binds field |
| | \| *opcode* ( *arguments* ) | Constructor application |
| | \| *pattern pat-op pattern* | Conjoins, concatenates, disjoins patterns |
| | \| `some` *class-name* | Matches a single token of the class named |
| | \| *label-name* : *pattern* | Labels pattern |
| | \| *pattern* ... | Less restrictive conjunction |
| | \| ... *pattern* | Less restrictive conjunction |

Patterns and their composition are most easily understood by looking at the rules for matching patterns. Patterns are tested for matching against sequences of tokens; the special pattern `epsilon` matches the empty sequence. For each constraint, the toolkit checks the field named therein to see if it falls in the range specified in a range constraint or is equal to the value bound in a field binding.

Patterns can be combined by conjunction, concatenation, or disjunction. When `p` and `q` are patterns, a conjunction "`p & q`" matches if both `p` and `q` match. We typically use conjunction to constrain multiple fields within a single token. A concatenation "`p; q`" matches if `p` matches an initial sequence of tokens and if `q` matches the following tokens. We typically use concatenation to build up patterns matching sequences of more than one token, for example, to match effective addresses on the Pentium. A disjunction "`p | q`" matches if either `p` or `q` matches. We typically use disjunction to group patterns for instructions that are related, e.g., to group the SPARC integer-arithmetic instructions.

The wildcard constraint "`some` *class*" matches any token of class *class*; for example, on the SPARC, "`some itoken`" matches any 32-bit token.

The labeled pattern `L: p` matches whenever `p` matches, and it binds the identifier `L` to the location in the instruction stream where `p` matches.

The ellipsis has no effect on matching, but it relaxes restrictions on conjunction, as described below.

Patterns solve the intellectual problem of describing binary representations. Each composition operator addresses a different need. Conjunction specifies how values in fields are combined to form tokens. Concatenation describes representations containing multiple tokens in sequence. Disjunction describes alternatives. Concatenation and disjunction operators are found in regular expressions. Unlike regular expressions, patterns do not have a Kleene closure (repetition) operator. This omission, together with the ability to examine fields in any order,

| *Component* | | *Contains... combined with* | | *Matches when...* |
|---|---|---|---|---|
| Pattern | (disjunction) | disjuncts | \| | any disjunct matches |
| Disjunct | (sequence) | sequents | ; | each sequent matches a token |
| Sequent | (conjunction) | constraints | & | token satisfies every constraint |
| Range constraint | | $lo \leq f < hi$ | | field falls within range |
| Field binding | | $f = expression$ | | always |

Fig. 1. Some components of the normal form of patterns.

distinguishes the problem of matching patterns from the problem of matching regular expressions.

*Representing Patterns.* This section presents a detailed description of the toolkit's representation of patterns. Studying the details of the representation is the best way to understand the meanings of patterns and the pattern operators and to understand the utility of patterns in generating encoders and decoders. The details can be confusing, because we use similar but not identical list structures at several levels, and because the structures play different roles in different contexts. Suggestive terminology helps distinguish structures and roles at each level.

Patterns are represented in a disjunctive normal form. The normal form has a three-level structure; the levels correspond to the three ways to combine patterns. Figure 1 shows the components of the normal form, the terminology used to refer to them, and the rules for matching them. We use several synonyms for each component, changing synonyms as we shift our focus from the component's role on its own to the component's relationship with the component above.

Every pattern is represented as a disjunction, that is, a list of alternatives. An empty list is permitted, even though the empty disjunction never matches.[1] Each disjunct, or alternative, is a sequence. Each item in a sequence is a conjunction of constraints. A pattern matches a sequence of tokens when one of its disjuncts (alternatives) matches. That disjunct matches a sequence of tokens when every sequent (conjunction) matches the corresponding token. The empty sequence, denoted by `epsilon`, always matches, consuming no tokens. Finally, a conjunction matches a token if the token satisfies all of the constraints in the conjunction. Each conjunction applies to a particular class of tokens, and all the constraints in the conjunction must constrain fields from that class. The empty conjunction, which is denoted by `some` *class*, is permitted; it matches any token of the associated class.

We define the *shape* of a sequence to be the list of token classes associated with the conjunctions of that sequence. Encoding and decoding choose a particular disjunct (sequence) to emit or match, and the shape of the sequence determines which tokens are emitted or matched when that sequence is encoded or decoded.

We can define simple constraints and the pattern operators in terms of the normal form of patterns. It is not hard to show that these definitions, combined with the rules for matching in normal form, imply the matching properties described above.

The normal form of a simple constraint is a pattern with a single disjunct, which is a sequence of length 1, in which the single sequent contains the constraint. (A wildcard constraint has a form in which the sequent contains no constraints, i.e.,

[1] One can obtain an empty disjunction by conjoining mutually exclusive constraints.

it is the empty conjunction.) The normal forms of `p | q` and `p; q` are straightforward. We form `p | q` by concatenating the disjuncts of `p` and `q` to form one large disjunction. We form `p; q` by distributing concatenation over disjunction; and we concatenate two sequences by concatenating their sequents.

We also form `p & q` by distributing over disjunction, but the rules for conjoining two sequences are more complicated. The basic rule is that the sequences to be conjoined must have the same shape, i.e., they must be the same length, and the classes associated with corresponding sequents must be the same. For example, all of the conjunctions in the SPARC example operate on sequences of length 1, and each sequent comes from the `itoken` class. The Pentium is more complicated. For example, the pattern `mod = 0 & r_m = 5` is permitted, because both conjuncts constrain fields from the `ModRM` class. The pattern `mod = 0 & index = 2` is not permitted, because `mod` is from the `ModRM` class, but `index` is from the `SIB` class. We conjoin two sequences of identical shape by conjoining their individual sequents, elementwise. Conjoining two sequents simply means conjoining their constraints; if both sequents constrain the same field, their conjunction constrains the field to lie in the intersection of the two ranges.

The basic rule for conjunction is too restrictive on a machine like the Pentium, in which effective addresses of varying shapes must be conjoined with opcodes of a fixed shape. If the shape of one sequence is a prefix of the shape of another, we can conjoin two sequences elementwise until we run out of elements in the shorter sequence, and then we can take the remaining elements from the longer sequence unmodified. A similar technique works when one sequence is a suffix of another.

If the toolkit used prefixes or suffixes automatically, it might silently accept an unintended, incorrect conjunction, so it uses them only when told to do so. The specification writer uses an ellipsis ("`...`") before or after any pattern to liberalize conjunctions with that pattern. The pattern `p & q ...` is defined whenever `q`'s shape is a prefix of `p`'s shape. `q` is conjoined with the prefix of `p` whose shape matches its shape, and the rest of `p` is concatenated to the result. Similarly, `p & ... q` is defined whenever `q`'s shape is a suffix of `p`'s shape, and the patterns are aligned at the end instead of the beginning. The ellipsis has the effect of making a pattern "lose its shape" where the ellipsis appears; so `p ... & ... q` is never legal, because `p ...` has no well-defined suffix, and `... q` has no well-defined prefix.

The restrictions on conjunction, with or without the ellipsis, guarantee that each disjunct in a valid pattern corresponds to a sequence of tokens. The toolkit uses this invariant to generate both encoders and decoders. These rules prohibit "mixing" tokens of different classes; in each instruction, each sequence of bits comes from a token of a unique class.

*Conditions and Names.* Free variables may appear not only in field bindings, but also in *conditions* associated with a pattern. No conditions appear in the grammar for patterns; instead, conditions are implicit in other parts of the specification and are associated with patterns in the toolkit's internal representation. For example, encoding of a field binding is subject to the condition that the computed value fit in the field; the condition becomes part of the pattern in which the field binding appears. Internally, this condition is derived from an operator that narrows a value to fit in the number of bits available. The toolkit uses a signed narrow for sign-

| *Component* | *Contains* | *Matches when...* | *To encode...* |
|---|---|---|---|
| Pattern | disjuncts, name | Any disjunct matches. | Encode first disjunct with satisfied conditions. |
| Disjunct | conditions, sequents+labels, ellipses, name | Conditions are satisfied, and each sequent matches. | Encode each sequent as one token. |
| Sequent | constraints, field bindings | Constraints and bindings match. | Set fields of token using constraints and bindings; emit token. |
| Label | | Always matches; binds identifier to location. | Not encoded, but may be used in equations. |
| Range constraint | $lo \le f < hi$, $lo$ & $hi$ constant | Field value falls in range. | If range has one element, set field. |
| Field binding | $f = expression$ | Always matches; equates expression to value of field. | Set field to value of expression. |

Fig. 2. Normal form of patterns, with matching and encoding rules.

extended fields and an unsigned narrow for other fields. From the unsigned narrow, the toolkit derives the condition $0 \le f < 2^n$, for a value $f$ put into a field of $n$ bits. From the signed narrow, the toolkit derives the condition $-2^{n-1} \le f < 2^{n-1}$. Other conditions may be derived from equations in a constructor definition. For example, most RISC branch instructions are described by equations that have solutions only under the condition that the target address differs from the program counter by a multiple of the word size.

We associate conditions with each disjunct. Although conditions could be associated with each constraint or each sequent, the disjunct is a better choice, because it is the largest component of a pattern that must be matched in its entirety. The disjunct is also the natural place to put conditions associated with constructor definitions. For example, the binary representation of a SPARC branch instruction is represented by a pattern of one disjunct; the disjunct includes the condition $(addr - L) \bmod 4 = 0$, where $L$ represents the location of the instruction, and $addr$ represents the target address of the branch instruction.

Both patterns and disjuncts have names. A pattern's name can be used wherever a pattern is expected. Disjunct names are used to compute constructors' names when patterns are used in constructor opcodes.

Figure 2 shows the full representation of patterns, together with the rules for matching and encoding them. As an example, the `alu` pattern from the SPARC specification has 38 disjuncts and the name `alu`. The first disjunct has no conditions, one sequent, no labels, no ellipses, and the name `add`. The single sequent of that disjunct is a sequent of class `itoken`. It has two range constraints, $2 \le \texttt{op} < 3$ and $0 \le \texttt{op3} < 1$, and no field bindings.

*Using and Naming Patterns.* Patterns are used in specifications in two ways. Opcodes are defined by binding names to pattern *values*, which contain no field bindings and are computed statically. Constructors and matching statements are defined using pattern *expressions*, which may contain free variables whose values are not known until encoding or decoding time. Such variables must be operands of the constructor; that is, they must be bound by the constructor's definition.

The `patterns` declaration binds names to pattern values; pattern expressions are used in constructor definitions and matching statements, which are described below. Pattern bindings are typically used to define opcodes and to group related opcodes, e.g., they are used to define the SPARC opcodes. Their syntax is

*spec* ⇒ `patterns` {*pattern-binding*}

*pattern-binding* ⇒ *pattern-name* `is` *pattern*
  | `[` {*pattern-name*} `]` `is` *pattern*
  | *pattern-name* `is any of [` {*pattern-name*}`],`
   `which is` *pattern*

Patterns bound to the special name "_" are ignored. Such patterns may correspond to unused opcodes, as in Table F-3 in the SPARC example. A pattern binding can bind one name to one pattern or each of a list of names to one of a list of patterns. Lists of patterns are created by using *generating expressions* in constraints. Generating expressions are modeled on expressions in Icon, which can produce more than one value [Griswold and Griswold 1990]. They are ranges or lists:

*generating-expression* ⇒ `{` *lo* `to` *hi* [`columns` *n* ]`}` | `[` {*integer*} `]`

The values generated are enumerated in left-to-right LIFO order. For example, the SPARC example's declaration for Table F-1 binds the names `TABLE_F2`, `call`, `TABLE_F3`, and `TABLE_F4` to the patterns `op = 0`, `op = 1`, `op = 2`, and `op = 3`, respectively.

### 3.3 Constructors

A constructor maps a list of operands to a pattern, which stands for the binary representation of an operand or instruction. Typed constructors produce operands; untyped constructors produce instructions. Because most manuals describe instructions in terms of their assembly-language syntax, we designed constructor specifications to resemble that syntax. A constructor specification begins with an opcode and a list of operands. It also gives a type and zero or more "branches," which designate possible representations.

*spec* ⇒ `constructors` {*constructor*}

*constructor* ⇒ *opcode* {*operand*} [ `:` *type-name* ] [*branches*]

A constructor without explicit branches is given the representation obtained by conjoining the opcode with the operands.

The type of a constructor determines how the corresponding encoding procedure can be used. Although a constructor with no explicit type is called "untyped," in fact it has a predefined, anonymous type—the type of instructions. Corresponding encoding procedures emit instructions. The encoding procedures for explicitly typed constructors produce values that can be used as operands to other constructors, as described below.

Opcodes are tricky. They can be simple strings, or they can be combinations of strings, pattern names, and field names, which are expanded to define multiple constructors with one specification. For example, the SPARC `alu` constructor specification expands the `alu` pattern to define 34 constructors at once. Compound opcodes are formed by joining strings or names using the ^ symbol.

*opcode* ⇒ *opname* {^ *opname*}

An *opname* can be the name of a field or pattern, or it can be an unbound name or a string. Unbound names mean the same as strings; for example, in the SPARC example, because the *opname* `dispA` is not previously defined, it is equivalent to `"dispA"`. This notational convenience means that the names of constructors seldom need to be quoted.

When any *opname* is the name of a pattern or field, the toolkit expands opcodes by enumerating the disjuncts of patterns and the named values of fields. For example, the toolkit expands the `branch^a` opcode by expanding the pattern `branch` to the 16 disjuncts named in its definitions, and it expands the field `a` to the two named values `""` and `",a"`. The SPARC example's single constructor definition of `branch^a` is therefore equivalent to a series of 32 definitions:

```
constructors
  "bn"    addr { addr = L + 4 * disp22! } is  L: bn  & disp22 & a = 0
  "bn,a"  addr { addr = L + 4 * disp22! } is  L: bn  & disp22 & a = 1
   ...
  "bvc,a" addr { addr = L + 4 * disp22! } is  L: bvc & disp22 & a = 1
```

Because architecture manuals often use the same name to refer both to an opcode and to its instruction, we put constructors in a separate name space, so the same name can be used to refer both to constructors and to patterns.

Operands may be fields, integers, or patterns. Field and integer operands may be signed or unsigned, and they may be designated relocatable. Pattern-valued operands must result from applying constructors of a designated type. Operand types are distinguished by their names; an operand is a field or pattern if its name is that of a field or a constructor type, and it is an integer otherwise.

The type of an operand determines how its name can be used on the right-hand side of a constructor. Integer operands can be used only in integer expressions, which appear in field bindings. Field operands can be used as integers, but they can also be used as patterns, in which case the field name stands for the pattern binding that field to the corresponding operand, as shown in the SPARC example. Finally, pattern-valued operands can be used only as patterns.

A list of operands may be decorated with spaces, commas, brackets, quoted strings, and other punctuation. The punctuation represents assembly-language syntax, and the toolkit uses it to generate encoding procedures that emit assembly language and to generate a grammar that recognizes assembly language.

Constructors solve several intellectual problems. They give an abstract structure to an instruction set, they connect that structure both to binary representations and to assembly language, and they formalize instructions as functions mapping operands to binary representations. An instruction set's abstract structure comes from the types of the constructors and their operands. This structure is isomorphic to a grammar in which the start nonterminal corresponds to the anonymous type "instruction," and in which each explicit constructor type corresponds to an additional nonterminal. Each constructor corresponds to a production in which the constructor's type appears on the left-hand side, and its operands appear on the right. The terminal symbols of the grammar are the operands that are fields, integers, or relocatable addresses. The patterns on the right-hand sides of constructor definitions are equivalent to synthesized attributes of the grammar. Field names, constructor names, and punctuation define an assembly-language representation that is implicit in every constructor definition, and these representations are also equivalent to synthesized attributes of the grammar.

Relocatable addresses are not essential to the intellectual task of specifying representations; instead, they support separate compilation in encoding applications. Any field or integer operand can be designated relocatable by

$$\mathit{spec} \Rightarrow \texttt{relocatable}\ \{\mathit{identifier}\}$$

For example, the `addr` operand of the SPARC `branch` constructor is declared relocatable. Labels that appear in constructors' patterns are also relocatable. Applications typically use relocatable addresses to refer to locations bound after encoding, at link time. Allowing any operand to be relocatable simplifies implementation of applications that usually emit assembly language. For example, it simplifies construction of `mld`'s code generators, because it enables automatic translation of existing assembly-emitting code generators into `mld`'s binary-emitting code generators. Without the ability to make any operand relocatable, large parts of `mld`'s code generators would have to be written by hand.

When a constructor that uses relocatable operands is applied, it checks to see if their values are known (e.g., they have been assigned absolute addresses). If so, it treats them as ordinary integers and emits the instruction. Otherwise, it emits placeholder patterns and creates a relocation closure [Ramsey 1996a]. The application holds the closure until the addresses on which it depends become known, at which point it applies the closure to overwrite the placeholder with the correct encoding of the instruction. Alternatively, the toolkit provides a machine-independent representation that can be used to write the closure to a file, from which another application could read and apply it.

Placeholder patterns are associated with token classes:

$$\mathit{spec} \Rightarrow \texttt{placeholder for}\ \mathit{class\text{-}name}\ \texttt{is}\ \mathit{pattern}$$

The toolkit uses the shape of a constructor's pattern to compute its placeholder, so the placeholder is the same size as the relocated instruction that will overwrite it.

The *branches* of a constructor specification contain equations and patterns. The patterns specify binary representations, and the equations relate the constructor's operands to the variables used in the patterns.

$$\begin{aligned} \mathit{branches} \Rightarrow\ & [\texttt{\{}\ \mathit{equations}\ \texttt{\}}]\ [\texttt{is}\ \mathit{pattern}] \\ |\ & \{\texttt{when \{}\ \mathit{equations}\ \texttt{\} is}\ \mathit{pattern}\ |\ \texttt{otherwise is}\ \mathit{pattern}\} \end{aligned}$$

When a constructor has a single branch, the pattern can be omitted, in which case it is taken to be the conjunction of the constructor's opcode with its operands. The ability to specify multiple branches supports conditional assembly, as with the SPARC `set` constructor. When encoding, the toolkit emits the first branch for which all conditions are satisfied. As explained above, conditions are satisfied when (1) all values bound to fields fit in those fields and (2) all equations used in the branch have solutions. When decoding, the toolkit matches any of the branches. `Otherwise` is syntactic sugar for `when { }`.

Equations express relationships between operands and fields. As written, they relate sums of terms with integer coefficients. Terms include field and integer variables, from which one can extract bits by `n@[lo:hi]`. One can also sign-extend a variable or extracted bits with the postfix exclamation point, as shown in the descriptions of the SPARC branch constructors. Equations may include inequalities, which become conditions attached to disjuncts of a branch's pattern. Conditions may also arise from solving equations; for example, the condition $(addr - L) \bmod 4 = 0$, which is attached to the patterns in the SPARC branch constructors, is derived from the equation for those constructors. All conditions must be satisfied for the constructor to be matched or encoded.

The toolkit uses a simple equation solver [Ramsey 1996b]. To encode, the toolkit takes operands as known and solves for fields. To decode, the toolkit takes fields as known and solves for operands.

Constructors are represented essentially as lambda terms mapping operands to patterns. The results of solving equations are represented in the patterns as conditions or as expressions in field bindings, so the only free variables in a constructor's pattern are the constructor's operands. Constructors with multiple branches, like the `set` constructor in the SPARC example, result in patterns with multiple disjuncts. The encoding procedure associated with the constructor emits the first disjunct whose conditions are known to be satisfied. If a condition depends on the value of an unknown relocatable operand, the toolkit conservatively assumes that the eventual value may not satisfy the condition, and it moves on to the next disjunct. If all disjuncts depend on relocatable operands, the toolkit uses the final disjunct. This technique, while safe, is unsuitable for emitting span-dependent instructions; for example, it uses the most general representation for all forward branches. We believe that standard techniques for resolving span-dependent instructions [Szymanski 1978] can be applied to our specifications.

### 3.4 Matching Statements and Decoding

Decoding applications use the toolkit's matching statements. Matching statements provide a notation for writing instruction recognizers that are efficient and easily understood. Matching statements resemble ordinary case statements, but their arms are labeled with patterns. The first arm whose pattern matches is executed. The syntax for matching statements is

$$\begin{aligned} \textit{matching-statement} \Rightarrow\ & \texttt{match}\ \textit{code}\ \texttt{to} \\ & \big\{ \texttt{|}\ \textit{pattern}\ \big[\texttt{\{}\ \textit{equations}\ \texttt{\}}\big]\ \big[\texttt{[}\ \textit{name}\ \texttt{]}\big]\ \texttt{=>}\ \textit{code} \big\} \\ & \big[\texttt{else}\ \textit{code}\big] \\ & \texttt{endmatch} \end{aligned}$$

The terminal symbol *code* stands for a fragment of Modula-3 or C code. The *code* next to `match` evaluates to a location in an instruction stream. The representation of the instruction stream is implicit in code templates supplied by the application writer, as described below. Each arm may include equations that must be satisfied for the arm to match. A name in square brackets is bound to the name of the

```
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;
```

Fig. 3. Matching statement used for control-flow analysis of SPARC instructions.

pattern that matched. If an arm's pattern matches, the *code* on the right-hand side of `=>` is executed.

*Matching-statement* is itself a grammatical start symbol; it cannot be derived from *specification*. When generating decoders, the toolkit's translator reads a *specification* from one file, then transforms a different file containing one or more *matching statements*.

In a matching statement, every free variable in a pattern is a binding instance; the toolkit computes a value for each such variable, and the values can be used in the host-language *code* on the right-hand side of the arm labeled by the pattern. Free variables associated with typed constructors are bound to locations in the instruction stream. The generated decoder converts such bound locations to integers.

Matching statements can be embedded in programs written in Modula-3 or in C. The toolkit's translator acts as a simple preprocessor—it finds embedded matching statements and rewrites them into pure Modula-3 or C code.

Matching statements make an application's decoding code clear and concise. For example, `ldb`, a retargetable debugger for ANSI C, uses matching statements to implement control-flow analysis. Most of `ldb`'s breakpoint implementation is machine independent; the only machine-dependent part is the analysis of control flow [Ramsey 1994a]. Figure 3 shows a simplified version of the SPARC code in `ldb`'s breakpoint implementation, omitting subtleties associated with delayed branches. This code finds which instructions could be executed immediately after an instruction at which a breakpoint has been planted [Ramsey 1994a]. After an ordinary instruction, the only instruction that can follow is its inline successor, as computed by the first arm of the matching statement. `FollowSet.T{L}` is a set of addresses containing the single element `L`, which is the location of the successor instruction. Calls and unconditional branches also have only one instruction in their "follow set," but conditional branches have two. The two `jmpl` patterns are indirect jumps through registers; the `GetReg` procedure gets the value in the register in order to compute the target address. The matching statement in Figure 3 expands to nested case statements totaling about 90 lines of Modula-3 code. The count does not convey the difficulty of writing the code by hand, because the toolkit eliminates unnecessary tests by combining seemingly unrelated opcodes if they result in execution of the same code.

Application writers can use any representation of instruction streams; in particular, the toolkit does not constrain the application to use integers to represent locations. An application writer specifies a representation by supplying the toolkit with four code fragments: the data type used to represent locations, a template used to add an integer offset to a location, a template used to convert a location to an unsigned integer, and a template used to fetch a token of a specified width from a location. The templates are specified by

$$\begin{aligned} \textit{spec} \Rightarrow\ & \texttt{fetch}\ (\textit{width} \mid \texttt{any})\ \texttt{using}\ \textit{template} \\ \mid\ & \texttt{address type is}\ \textit{template} \\ \mid\ & \texttt{address add using}\ \textit{template} \\ \mid\ & \texttt{address to integer using}\ \textit{template} \end{aligned}$$

The *template* symbols stand for quoted strings containing fragments of Modula-3 or C code mixed with escape sequences that stand for addresses, widths, and offsets. Widths are measured in bits; offsets are measured in units of `pc_unit_bits`:

$$\textit{spec} \Rightarrow \texttt{pc\_unit\_bits}\ \textit{width}$$

This size must evenly divide the width of every token; the default size is 8 bits.

The toolkit builds a decision tree for each matching statement. The decision tree checks all applicable range constraints while examining each field at most once. If patterns in two arms use the same range constraints but have different conditions, the toolkit checks conditions sequentially, but this situation is rare. The toolkit tries to minimize the number of tests needed to identify an arm. No polynomial-time algorithm is known for this problem, and even though the toolkit builds decision trees at tool-compile time, it would take too long to generate and evaluate all possible decision trees. Our heuristics yield trees that are at least as good as trees we would write by hand.

## 4. SPECIFYING CISC INSTRUCTIONS

Tools may work well for RISC architectures without being very useful for CISC architectures. To demonstrate the utility of our specification language, we show two complex aspects of our Pentium specification: addressing modes and variable-sized operands. Figure 4 shows constructor specifications for the Pentium's addressing modes. We have given each constructor the type `Eaddr`, which we have chosen to represent effective addresses. Values of type `Eaddr` are used as operands to untyped constructors, as shown below. Again, the brackets and asterisks in the specification are punctuation indicating suggested assembly-language syntax. Figure 5 depicts the structures of the patterns used in Figure 4.

Effective addresses begin with a one-byte ModR/M token, which contains an addressing mode and a register. In indexed modes, the ModR/M token is followed by a one-byte SIB token, which holds `index` and `base` registers and a scale factor `ss`.

```
constructors
 Reg        reg  : Eaddr                          is mod = 3 & r_m = reg
 Indir     [reg] : Eaddr {reg != 4, reg != 5} is mod = 0 & r_m = reg
 Disp8   d![reg] : Eaddr {reg != 4, d = i8! } is mod = 1 & r_m = reg; i8
 Disp32  d![reg] : Eaddr {reg != 4}           is mod = 2 & r_m = reg; i32 = d
 Abs32   a       : Eaddr                          is mod = 0 & r_m = 5;   i32 = a
constructors
 Index     [base][index * ss] : Eaddr { index != 4, base != 5 } is
                      mod = 0 & r_m = 4; index & base     & ss
 Index8  d![base][index * ss] : Eaddr { index != 4, d = i8!   } is
                      mod = 1 & r_m = 4; index & base     & ss; i8
 Index32 d![base][index * ss] : Eaddr { index != 4 } is
                      mod = 2 & r_m = 4; index & base     & ss; i32 = d
 ShortIndex    d![index * ss] : Eaddr { index != 4 } is
                      mod = 0 & r_m = 4; index & base = 5 & ss; i32 = d
```

Fig. 4. Constructor definitions for the Pentium's 32-bit addressing modes.

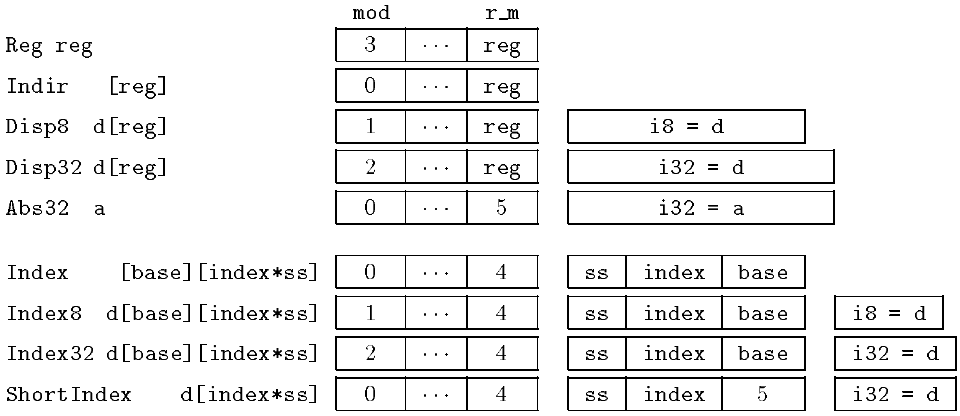


Fig. 5. Tokens used in the Pentium's 32-bit addressing modes. Token sizes are not to scale.

Finally, some modes take immediate displacements [Intel Corporation 1993, Tables 26-2 to 26-4]. The tokens and fields used in effective addresses are as follows:

```
fields of ModRM (8)  mod 6:7 reg_opcode 3:5 r_m  0:2
fields of SIB   (8)  ss  6:7 index      3:5 base 0:2
fields of I8    (8)  i8  0:7
fields of I16  (16)  i16 0:15
fields of I32  (32)  i32 0:31
```

The fields `i8`, `i16`, and `i32` occupy whole tokens.

We define constructors of type `Eaddr` to create effective addresses in 32-bit mode. The first group of constructors specifies the nonindexed addressing modes. The simplest mode is encoded by `mod = 3`; it is a register-direct mode that can refer to any of the machine's 8 general registers. The next 3 modes are register-indirect modes with no displacement, 8-bit displacement, and 32-bit displacement. The 8-bit displacement is computed by sign-extending the `i8` field. Semicolons separate ModR/M tokens from the displacement tokens that follow.

The inequality `reg != 5` shows that `r_m` may not take the value 5 in simple indirect mode. Instead of denoting indirect use of the base pointer, which is the register normally encoded by 5, the combination `mod = 0 & r_m = 5` encodes a 32-bit absolute mode. The inequality `reg != 4` shows that the value 4 may not be used to encode indirect use of the stack pointer, which is the register normally encoded by 4. This value is used instead to encode the indexed modes, which use an SIB token as well as the ModR/M token.

The indexed modes are the second group in Figures 4 and 5. The ModR/M token in which `r_m = 4` is followed by an SIB token. The stack pointer may not be used as an index register (`index != 4`). Depending on the value of `mod` in the ModR/M token, the SIB token may end the address, or an 8-bit or 32-bit displacement may follow. Finally, "`mod = 0 & base = 5`" denotes an indexed address with no base register and a 32-bit displacement.

None of the addressing modes specifies a value for the `reg_opcode` (middle) field of the ModR/M token. This field is not part of the effective address; depending on the instruction, it can be part of the opcode, or it can denote a register operand. Effective addresses are used by conjoining them with a pattern that constrains `reg_opcode`; the resulting pattern specifies every bit of the ModR/M token. We need the ellipsis operator to make the conjunction work. Even though effective addresses have several different shapes, all the shapes begin with `ModRM`, so it is legal to write `Eaddr & p ...` whenever `p`'s shape is `ModRM`. The move-byte and move-byte-immediate instructions show the use of the ellipsis:

```
constructors
  MOV^"mrb" Eaddr, reg is MOV & Eb.Gb; Eaddr & reg_opcode = reg ...
  MOV.Eb.Ib Eaddr, i8! is MOV.Eb.Ib;   Eaddr & reg_opcode = 0 ...; i8
```

Our specifications of the Pentium's opcodes, which are not shown in this article, mimic the tables in the manual [Intel Corporation 1993]. The manual uses families of opcodes (`ADD`, `MOV`, etc.) that are distinguished by suffixes indicating the locations and sizes of the destination and source operands. The suffix "Eb,Gb" indicates that the destination is given by an effective address, that the source is in a general-purpose register, and that both source and destination operand are one byte wide. In many cases, as with "MOV Eb,Gb", we specify the operation and the suffix separately, then conjoin them to get an opcode, thereby writing $m+n$ specifications instead of $m \times n$ specifications. The "Eb,Ib" suffix, which uses an immediate operand as the source, cannot use this scheme, so we specify the full opcode as `MOV.Eb.Ib`.

The Pentium uses an unusual method of identifying the sizes of operands. Most instructions come in three variants: one each for 8-bit, 16-bit, and 32-bit operands. Typically the 8-bit variant has a distinct opcode, but the 16- and 32-bit variants share an opcode and are distinguished by the presence or absence of an instruction prefix. We specify an "object varying" pattern as a sequence that is empty or that contains the prefix

```
patterns ow is OpPrefix
         od is epsilon
         ov is ow | od
```

where `ow` is mnemonic for "object word" and `od` for "object doubleword." This specification assumes that the hardware codes for 32-bit doubleword operands by default; the alternate assumption could be specified by exchanging the definitions of `od` and `ow`. To specify both the 16- and 32-bit variants of the memory-to-register move instruction, we write

```
constructors
  MOV^"mr"^ov Eaddr, reg is ov; MOV & Ev.Gv; Eaddr & reg_opcode = reg ...
```

This specification differs from the move-byte specification in that we have used the suffix "Ev,Gv", which codes for operands of either word or longword ("variable") size, depending on the presence or absence of a prefix. The pattern `ov` expands to the prefix for the 16-bit variant and to the empty sequence for the 32-bit variant.

When immediate operands are used, all three variants must have separate specifications, because the operands are different sizes. The 8-bit move-immediate instruction appears above; the remaining variants are specified by

```
constructors
  MOV.Eb.Iv^ow Eaddr, i16! is
        ow; MOV.Ev.Iv; Eaddr & reg_opcode = 0 ...; i16
  MOV.Eb.Iv^od Eaddr, i32! is
        od; MOV.Ev.Iv; Eaddr & reg_opcode = 0 ...; i32
```

Again, only one of these instructions has a prefix, since `od` stands for the empty sequence.

Two features of SLED exist only to enable the description of CISC machines. One, the ability to define tokens of different sizes and classes, is used only to describe the Pentium and the Motorola 68000. The other, the ability to form sequences of tokens, is used in both CISC and RISC specifications, but we have used it only rarely in RISC specifications, typically to synthesize "instructions" from multi-instruction sequences.

Owen C. Braun's description of the 68000 [Braun 1996] exposes several shortcomings of SLED. Some addressing modes have different representations, depending on where they are used; currently, they must be associated with distinct sets of constructors of distinct types. For example, a compiler writer must call one of two procedures to encode a register-direct mode, depending on whether it is to be the source or the destination operand of a move instruction. Not all of the 68000's addressing modes are valid in all instructions; there are several different subsets, such as the "data-alterable" modes, for example. Our (incomplete) specification of the DSP56000 exhibits similar problems. These problems can be handled by defining multiple sets of constructors, but the resulting specifications are ugly and difficult to maintain.

We are considering two extensions that would help improve specifications of the 68000 and the DSP56000 and would help specify address prefixes on the Pentium. One would enable us to attach multiple pattern-valued attributes to constructors and to use different attributes to specify alternate representations or parts of representations. Another would support simple specification of subsets of typed constructors, which we could use to specify restrictions on addressing modes. In both cases, we believe that simplifications in CISC specifications will justify the extra complexity in SLED. Because we have not implemented these extensions, we consider the details beyond the scope of this article.

## 5. IMPLEMENTATION

The toolkit's translator, generator, and checker are combined in a single Icon program [Griswold and Griswold 1990] of about 10,000 lines. We omit the details of the implementation, but we do explain what the implementation does, what it assumes, and how the toolkit's library supports those assumptions.

For each matching statement, the toolkit generates an efficient decoder using nested case statements. These decoders manipulate instruction streams using the code templates supplied by the application writer. Because the decoders need only what is in the templates, they are isolated from other properties of the decoding application, including byte order. They are also independent of any generated encoders and of the toolkit's library.

The toolkit creates an encoding procedure from each constructor in a specification. Procedures that come from typed constructors are useful only for producing operands to be passed to other encoding procedures. In particular, such procedures never have side effects; they return values. Procedures generated from untyped constructors do have side effects; they emit instructions. If the constructor's pattern has no disjunct whose conditions are satisfied, the encoding procedure calls an error handler supplied by the application. Here are signatures for the C procedures that are generated from the **`Address`** constructor **`dispA`** and the untyped constructor **`ldsb`**, which appear in the SPARC example:

```
Address_Instance dispA(unsigned rs1, int simm13);
void ldsb(Address_Instance Address, unsigned rd);
```

The result of **`dispA`** could be used as the first argument to **`ldsb`**.

Normal encoding procedures emit binary representations, as determined by the encoding rules in Figure 2. The toolkit can also generate "encoding" procedures that emit assembly language. The assembly language is usually inferred from punctuation in constructor specifications, but it is possible to specify assembly syntax separately, as described in the toolkit's reference manual [Ramsey and Fernández 1994b]. This ability is useful when several assembly languages are in common use for a single architecture, as is the case for the Pentium.

The toolkit can generate direct or indirect interfaces to encoding procedures. Indirect interfaces use interface records—structures containing function pointers. Applications can choose binary or assembly language at run time by using a pointer to the appropriate interface record.

Binary encoding procedures have side effects on a global instruction stream. When values of relocatable operands are not available, they also create relocation information in the form of closures. The encoding procedures make certain assumptions about instruction streams and relocatable operands. Here we enumerate the assumptions and explain the implementations in the toolkit's library, which satisfy the assumptions.

A *relocatable address* represents the value of a relocatable operand. It is an abstraction with two operations: *force* and *known*. *Force* takes a relocatable address and produces an (integer) absolute address. *Known* tells whether *force* can be applied. Generated encoding procedures use *known* to decide whether to emit tokens or to create relocation closures, and they use *force* to get the values of the operands themselves.

An *instruction stream* holds tokens emitted by encoding procedures. It has a *location counter* that marks the location of the next token to be emitted. Like relocatable addresses, the location counter supports the *known* and *force* operations. Encoding procedures assume that they can manipulate the location counter and that they can call *emitters* to put tokens into the instruction stream. Emitters write bits and advance the location counter. The library includes a little-endian emitter, a big-endian emitter, and two emitters that use the native byte order of the host machine, or an application can supply its own emitters. One of the native emitters is faster than the other, but it requires that the location counter always be aligned on a multiple of the token size.

Most encoding applications need a richer model of instruction stream than that assumed by the toolkit's encoding procedures. The toolkit's library provides *relocatable blocks*, which implement the instruction-stream abstraction. They also support many other operations, including changing blocks and locations, assigning addresses to blocks, emitting tokens into blocks, and writing blocks into files or memory. An application can use any number of relocatable blocks, and it can emit tokens into a block before the block's address has been assigned. For example, a UNIX assembler might use three blocks, one each for the code, initialized data, and uninitialized data sections. The assembler would let the linker determine and assign the addresses of those blocks.

A *label*, which points to a location in a relocatable block, provides the basic *known* and *force* operations.[2] The toolkit does not associate names with labels; applications can use any method to name and find labels. For more flexibility, the library also provides an implementation of relocatable addresses that represents an address as the sum of a label and a signed offset. This representation is adequate for applications like compilers and linkers. Authors of other applications can use more sophisticated representations (e.g., linear expressions over addresses and labels) without changing the code generated by the toolkit.

The toolkit needs little support from applications. Applications' primary obligations are to manage memory and to supply or select code for fetching and storing tokens. Encoding applications must supply a routine that the library uses to allocate memory for closures, labels, and relocatable blocks. Saving, applying, writing, and discarding closures are also the application's responsibility. In return, the application can choose its own policies for allocating memory and for managing closures. The toolkit is careful not to require large blocks of contiguous memory, not even to store large relocatable blocks. Finally, the toolkit provides no code to associate names with relocatable blocks, labels, or other abstractions; applications must supply their own.

The toolkit generates efficient code. When safety checks are elided, each encoding procedure executes about a dozen instructions. Generated decoders test each field at most once, and they test them in an order that quickly identifies the right arm of the matching statement.

The toolkit's generator can detect many internal inconsistencies in specifications, but it cannot identify specifications that are internally consistent but do not match

[2]This "label" is different from the labels introduced by the `L: p` construct, although both kinds serve the same function.

a target machine. There are several ways to write such incorrect specifications, for example, by getting operand order wrong or by interchanging names in an opcode table. The toolkit's checker [Fernández and Ramsey 1996] finds inconsistencies between the mapping specified in SLED and the mapping implemented by a trusted, independent assembler. The checker exploits the generator's ability to create encoding procedures for both binary and assembly representations. It exercises each constructor at least once, emitting both representations. The trusted assembler translates the assembly into binary, and the checker compares the two binary representations. If they are identical, the toolkit's specification is probably consistent with the assembler; if not, the toolkit and the assembler encode some instruction differently, and there is probably an error in the specification. A disassembler, which can be generated by the toolkit, makes it easier to find the source of the error.

## 6. RELATED WORK

Ferguson [1966] describes the "meta-assembler," which creates assemblers for new architectures. A meta-assembler works not from a declarative machine description but from macros that pack fields into words and emit them; it is essentially a macro processor with bit-manipulation operators and special support for different integer representations.

Most architecture-description languages emphasize the instruction semantics necessary for building tools that verify and simulate an instruction set, not the encoding and decoding descriptions necessary for building tools that process machine code.

Wick [1975] describes a tool that generates assemblers based on descriptions written in a modified form of ISP [Bell and Newell 1971]. His work investigates a different part of the design space; his machine descriptions are complex and comprehensive. For example, they describe machine organization (e.g., registers) and instruction semantics as well as instruction encoding.

LISAS [Cook and Harcourt 1994] is another specification language that includes distinct semantic and syntactic descriptions. It specifies binary representations by mapping sequences of named fields onto sequence of bits, a technique that works well for RISC machines, but is awkward for CISC.

The nML specification language [Fauth et al. 1995] uses an attribute grammar to specify instruction sets. The underlying grammar, without attributes, should be the same as the grammar induced by our constructors and their types. For specification, nML uses "OR-rules" and "AND-rules." The OR-rules are sums. They correspond to our constructor types when viewed as disjoint unions, and they also correspond to alternatives in a grammar. The AND-rules are products. They correspond to Cartesian products of operands of our constructors, and they also correspond to sequences of symbols in a production of a grammar.

nML and SLED use different notations and types to associate information with instruction sets. nML uses synthesized attributes to represent register-transfer semantics, assembly-language syntax, and binary representations. Writers can introduce extra attributes to represent things like addressing modes. The values of attributes may be integers, character strings, bit strings, or "register-transfer sequences." Binary representations are represented as bit strings. Attribute values are specified by writing explicit attribute equations for every production in the grammar, and they can be computed using C-like arithmetic functions, a `printf`-like formatting function, and a special notation for register-transfer sequences. An nML description can be used to build a simulator, which includes an instruction decoder, and a code generator, which includes a binary encoder. Using nML attribute equations to build an encoder appears straightforward, but the authors seem not to have published a description of how they invert the equations to produce a decoder.

SLED provides a more concise and less error-prone way of specifying binary representations than nML's binary-string attributes. SLED's generating expressions and constructor opcodes make it easy to specify many representations with few integer literals. Using patterns instead of bit strings relieves the specification writer from having to get the fields in the right order, and it helps the toolkit detect missing and duplicate fields. Finally, SLED specifications resemble architecture manuals; nML specifications do not. Our ideas could be exploited in the nML framework by including the pattern sublanguage (tokens, fields, and patterns) in nML and using pattern-valued attributes to specify binary representations. Conversely, nML's ideas could be exploited in our framework by adding nML's register-transfer sublanguage and by permitting users to attach arbitrary attributes to constructors and their operands. We expect that named, pattern-valued attributes would help users describe machines like the 68000 and DSP56000.

The GNU assembler provides assembly and disassembly for many targets, but different techniques are applied ad hoc to support different architectures [Elsner et al. 1993]. For example, Pentium instructions are recognized by hand-written C code, but MIPS instructions are recognized by selecting a mask and a sample from a table, applying the mask to the word in question, then comparing the result against the sample. On both targets, operands are recognized by short programs written for abstract machines, but a different abstract machine is used for each target. Another set of abstract machines is used to encode instructions during assembly. The implementations of the abstract machines contain magic numbers and hand-written bit operations. The programs interpreted by the abstract machines are represented as strings, and they appear to have been written by hand.

Larus and Schnarr [1995] use a machine description related to ours to provide machine-independent primitives that query instructions. The syntactic part of their machine description is derived from a subset of our language having only fields and patterns. They have added semantic information by associating register-transfer semantics with particular pattern names. From this combined syntactic and semantic information, the *spawn* tool generates classifiers that put instructions into categories like jump, call, store, invalid, etc. It finds the registers that each instruction reads and writes, and it generates C++ code to replicate such computations as finding target addresses. The descriptions used by *spawn* are both more and less powerful than ours. The semantic information makes it possible to derive a variety of predicates and transformations that are indispensable for instrumenting object code. The limited syntactic specification assumes there is only a single token (the "current instruction"), and it has no notion comparable to constructor, which makes it more difficult to understand how specifications are factored. Finally, *spawn* descriptions do not support encoding; instrumenters must provide preencoded "snippets" of machine code. The encoding is done by standalone compilers or assemblers, and the snippets are extracted from the resulting object code.

In spirit, our work is like ASN.1 [International Organization for Standardization 1987], which is used to create symbolic descriptions of messages in network protocols, but there are many differences. ASN.1 data can be encoded in more than one way, and in principle, writers of ASN.1 specifications are uninterested in the details of the encoding. ASN.1 encodings are byte-level, not bit-level encodings; ASN.1 contains an "escape hatch" (OCTET STRING) for strings of bytes in which individual bits may represent different values. Finally, ASN.1 is far more complex than our language; for example, it contains constructs that represent structured values like sequences, records, and unions, that describe optional, default, or required elements of messages, and that distinguish between tagged and "implicit" encodings of data.

## 7. EVALUATION

For code generation in traditional compilers, the toolkit is somewhat less suitable than a vendor's assembler. The toolkit does not easily support standard, machine-dependent formats for relocatable object code, and it does not provide optimizations that vendors may build into assemblers, like MIPS instruction scheduling.

SLED evolved from a simpler language used to recognize RISC instructions in a retargetable debugger [Ramsey 1992, Appendix B]. That language had field constraints and patterns built with conjunction and disjunction, but no concatenation and no constructors. There was no notion of instruction stream; instructions were values that fit in a machine word. We extended that language to specify encoding procedures by writing a constructor name and a list of field operands to be conjoined. This extension sufficed to describe all of the MIPS and most of the SPARC, and we used it to generate encoding procedures for `mld`. It could not, however, describe all of the SPARC, and it was completely unable to describe the Pentium, even after we added concatenation to the pattern operators. Two changes solved all our problems: making patterns explicit on the right-hand sides of constructor specifications and using constructor types to permit patterns as operands. We then realized there was no reason to restrict constructors to specifying encoding procedures, so we made it possible to apply constructors both in pattern definitions and in matching statements, yielding SLED as described in this article.

Patterns are a simple yet powerful way to describe binary representations. Field constraints, conjunction, and concatenation are all found in architecture manuals, and together they can describe any instruction on any of the four machines we have specified, as well as four other machines whose specifications are incomplete or have been written by our users. Patterns are not limited to traditional instruction sets in which opcode and operand are clearly separated; the machines we have described use instruction formats in which opcode bits are scattered throughout the instruction. Disjunction does not make it possible to specify new instructions, but it improves specifications by making it possible to combine descriptions of related instructions. By removing the need to specify each instruction individually, disjunction eliminates a potential source of error.

Constructor specifications provide clean, abstract representations of instructions and their operands, and they connect these abstractions to binary representations and to assembly language. Equations, though seldom used, are needed to describe instructions like relative branches, whose assembly-level operands differ from their machine-level fields. Equations can also express restrictions on operands, which are part of the definitions of some architectures, like the Intel Pentium.

We maximize SLED's expressive power by minimizing restrictions on the way patterns, constructors, and equations can be combined. For example, patterns and constructors can be used in each other's definitions, which makes it possible to factor complex architectures like the Pentium. Equations in constructor specifications are used for both encoding and decoding, and equations can also be used in matching statements. Because the elements of SLED work together, it is hard to see how the language could be simplified without destroying it. The simplicity of the specifications and the checking done by the toolkit combine to give users confidence in the correctness of the generated code.

## 8. AVAILABILITY

Version 0.5 of the toolkit implements SLED as described in this article, except that integer operands of constructors are always signed. It is available by anonymous ftp from ftp.cs.princeton.edu in directory pub/toolkit. The toolkit also has a home page at http://www.cs.princeton.edu/software/toolkit.

## 9. PRODUCTION NOTE

We prepared this article using the `noweb` tools for literate programming [Ramsey 1994b]. The examples have been extracted from this article and run through the toolkit, and they work with version 0.5.

### ACKNOWLEDGEMENTS

The editor and anonymous referees suggested a restructuring that helped improve the article. We are especially grateful for Referee 2's thorough reading and pointers to related work.

### REFERENCES

BALL, T. AND LARUS, J. R. 1992. Optimally profiling and tracing programs. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*. Albuquerque, NM, 59–70.

BELL, C. G. AND NEWELL, A. 1971. *Computer Structures: Readings and Examples*. McGraw-Hill, New York.

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

CATTELL, R. G. G. 1980. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems 2,* 2 (Apr.), 173–190.

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 128–137.

COOK, T. AND HARCOURT, E. 1994. A functional specification language for instruction set architectures. In *Proceedings of the 1994 International Conference on Computer Languages*. 11–19.

DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. 1996. Vortex: An optimizing compiler for object-oriented languages. *OOPSLA '96 Conference Proceedings,* in *SIGPLAN Notices 31,* 10 (Oct.), 83–100.

ELSNER, D., FENLASON, J., ET AL. 1993. *Using* `as`*: the GNU Assembler*. Free Software Foundation.

FAUTH, A., PRAET, J. V., AND FREERICKS, M. 1995. Describing instruction set processors using nML. In *The European Design and Test Conference*. 503–507.

FERNÁNDEZ, M. F. AND RAMSEY, N. 1996. Automatic checking of instruction specifications. Tech. rep., AT&T Research. July. Submitted to ICSE 1997.

FERGUSON, D. E. 1966. The evolution of the meta-assembly program. *Communications of the ACM 9,* 3, 190–193.

FERNÁNDEZ, M. F. 1995. 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 (June), 103–115.

GEORGE, L., GUILLAME, F., AND REPPY, J. H. 1994. A portable and optimizing back end for the SML/NJ compiler. In *5th International Conference on Compiler Construction.* 83–97.

GRAHAM, S. L., LUCCO, S., AND WAHBE, R. 1995. Adaptable binary programs. In *Proceedings of the 1995 USENIX Technical Conference.* New Orleans, LA, 315–325.

GRISWOLD, R. E. AND GRISWOLD, M. T. 1990. *The Icon Programming Language*, Second ed. Prentice Hall, Englewood Cliffs, NJ.

HASTINGS, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference.* San Francisco, CA, 125–136.

Intel Corporation 1993. *Architecture and Programming Manual.* Intel Corporation, Mount Prospect, IL.

International Organization for Standardization 1987. *Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1).* International Organization for Standardization. ISO 8824 (CCITT X.208).

JOHNSON, S. C. 1990. Postloading for fun and profit. In *Proceedings of the Winter USENIX Conference.* 325–330.

LARUS, J. R. AND SCHNARR, E. 1995. EEL: machine-independent executable editing. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices 30,* 6 (June), 291–300.

NELSON, G., Ed. 1991. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ.

RAMSEY, N. 1992. A retargetable debugger. Ph.D. thesis, Princeton University, Department of Computer Science. Also Technical Report CS-TR-403-92.

RAMSEY, N. 1994a. Correctness of trap-based breakpoint implementations. In *Proceedings of the 21st ACM Symposium on the Principles of Programming Languages.* Portland, OR, 15–24.

RAMSEY, N. 1994b. Literate programming simplified. *IEEE Software 11,* 5 (Sept.), 97–105.

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

RAMSEY, N. 1996b. A simple solver for linear equations containing nonlinear operators. *Software—Practice & Experience 26,* 4 (Apr.), 467–487.

RAMSEY, N. AND FERNÁNDEZ, M. F. 1994a. New Jersey Machine-Code Toolkit architecture specifications. Tech. Rep. TR-470-94, Department of Computer Science, Princeton University. Oct.

RAMSEY, N. AND FERNÁNDEZ, M. F. 1994b. New Jersey Machine-Code Toolkit reference manual. Tech. Rep. TR-471-94, Department of Computer Science, Princeton University. Oct.

RAMSEY, N. AND HANSON, D. R. 1992. A retargetable debugger. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices 27,* 7 (July), 22–31.

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

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices 29,* 6 (June), 196–205.

SRIVASTAVA, A. AND WALL, D. W. 1993. A practical system for intermodule code optimization. *Journal of Programming Languages 1,* 1–18. Also available as WRL Research Report 92/6, December 1992.

SZYMANSKI, T. G. 1978. Assembling code for machines with span-dependent instructions. *Communications of the ACM 21,* 4 (Apr.), 300–308.

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

Wick, J. D. 1975. Automatic generation of assemblers. Ph.D. thesis, Yale University.

Received December 1995; revised August 1996; accepted October 1996