1995

The New Jersey Machine-Code Toolkit

Norman Ramsey

Mary F. Fernández

Report Number:
95-082
THE NEW JERSEY MACHINE-CODE TOOLKIT

Norman Ramsey
Mary F. Fernandez

CSD TR-95-082
December 1995
The New Jersey Machine-Code Toolkit

Norman Ramsey, Purdue University*
Mary F. Fernández, Princeton University†

December 15, 1995

Abstract

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code. Applications that use the toolkit are written at an assembly-language level of abstraction, but they recognize and emit binary. Guided by a short instruction-set specification, the toolkit generates all the bit-manipulating code.

The toolkit’s specification language uses four elements: fields and tokens describe parts of instructions, patterns describe binary encodings of instructions or groups of instructions, and constructors map between the assembly-language and binary levels. These elements are suitable for describing both CISC and RISC machines; we have written specifications for the MIPS R3000, SPARC, and Intel Pentium instruction sets. By combining the elements in different ways, the toolkit enables machine-independent implementations of a number of machine-level concepts, including conditional assembly, span-dependent instructions, relocatable addresses, segments, object code, and relocation. The toolkit specifications can be checked automatically for consistency with existing standalone assemblers.

The implementation of the toolkit is largely determined by its representations of patterns and constructors. These elements are placed into a normal form that facilitates the construction of encoders. To build decoders, we rewrite the normal form to eliminate the ordering of tokens, and we then build a decision tree that examines parts of instructions in an order chosen to result in an efficient decoder.

We have used the toolkit to help write two applications: a retargetable debugger and a retargetable, optimizing linker. 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.

*1398 Computer Science Building, West Lafayette, IN 47907
†Author’s current address: AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
1 Introduction

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code—asmblers, disassemblers, code generators, tracers, profilers, and debuggers. The toolkit lets programmers encode and decode machine instructions symbolically. It transforms symbolic manipulations into bit manipulations, guided by a specification that defines mappings between symbolic and binary representations of instructions. We have written specifications for the MIPS R3000, SPARC, and Intel Pentium instruction sets. The specifications are simple, which makes it practical to use the toolkit to write applications for multiple architectures.

Traditional applications that process machine code include compilers, assemblers, linkers, and debuggers. Recent years have seen a host of new applications that process machine code, including 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, Guillame, and Reppy 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, Lucca, and Wahbe (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., most Unix compilers emit assembly language, not object code. It is not always practical to use an assembler, however—consider 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 can't be used as often as applications that work on object code, because source code is not always available. Our toolkit makes it easier to implement applications or frameworks that work with object code and can be used on any executable file.

Currently, applications that can't use an assembler 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.

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 labelled 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. The library implements both instruction streams and relocatable addresses, which refer to locations within the streams. The specification 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 toolkit's specification language is simple, and it is designed so that specifications can resemble instruction descriptions found in architecture manuals. It 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 inconsistencies. An instruction set can be specified with modest effort; our MIPS, SPARC, and Pentium specifications are 127, 193, and 460 lines.

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, the toolkit focuses on providing a high-level representation of instructions, to which semantic information can be coupled (see Section 5.1), and on automating the translation to and from that high-level representation.

We have 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. Idb, a retargetable debugger (Ramsey 1992; Ramsey and Hanson 1992), uses the toolkit to decode instructions and to implement breakpoints.

The toolkit provides practical benefits, like reducing retargeting effort. For example, 1db'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.
This paper makes several contributions. The toolkit solves only part of the retargeting problem, but it solves that part completely. The solution is both elegant and practical; the toolkit’s instruction-set specifications are clear, concise, and reusable, and the generated code is efficient. Our model of machine instructions makes several machine-level concepts general enough that they can be specified or implemented in a machine-independent way, including conditional assembly, span-dependent instructions, relocatable addresses, segments, object code, and relocation.

The rest of this paper is divided into four sections. The first and largest explains the elements of specifications and the toolkit’s models of relocation and matching. The next shows what the toolkit can do with specifications; it presents two applications built with the help of the toolkit, it shows how to check specifications for accuracy, and it tells the reader what to expect from the toolkit’s implementation. The penultimate section shows how specifications are used to create encoders and decoders; most of the techniques follow from our internal representation of the specification. The concluding section relates our work to other work, outlines some of the many problems we believe can be attacked using the ideas embodied in the toolkit, and evaluates the toolkit and its specification language.

2 Elements of Specifications and Applications

Because machine instructions don’t 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; 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. The toolkit lets the application writer choose a suitable byte order.

Each token in an instruction is partitioned into fields; a field is a contiguous range of bits within a token. Fields contain opcodes, operands, modes, or other information. Patterns constrain the values of fields; they may constrain fields in a single token or in a sequence of tokens. Simple patterns can be used to specify opcodes. More complex patterns can be used for such tasks as specifying the structure of addressing modes or defining the group of 3-operand arithmetic instructions.

Constructors connect the symbolic and binary representations of instructions. At a symbolic level, an instruction is an opcode (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. The result of the application is a pattern, which typically describes a sequence of tokens. Specification writers use constructors to define the equivalent of an assembly language. Application programmers use constructors to emit instructions, by calling procedures derived from constructor specifications, and to decode in-
structions, by using constructors in matching statements to match instructions and extract their operands.

Encoding applications often need to emit instructions before all the operands' values are known. The canonical example is the branch to an unknown label; the label's value may be determined on a second pass of an assembler, or it may not be available until link time. The toolkit's specification language permits any operand to be designated relocatable. Encoding procedures for constructors with relocatable operands are curried, so they can be applied separately to normal and relocatable operands. The first application emits a placeholder instruction and produces a closure, which is applied later to update the instruction once the values of the relocatable operands are known. Ramsey (1995a) describes a method of deriving closures that makes them nearly equivalent to the "relocation information" used in standard linkers.

Finally, decoding applications use patterns and constructors in matching statements, which provide a simple, readable way of writing code to recognize instructions, families of instructions, and sequences of instructions.

This section explains in detail the elements of the toolkit's specification language, its support for relocation, and its provision of matching statements. The following section shows how these specifications can be checked for correctness and used in applications.

2.1 Tokens and fields

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. The 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 get the value of a field in a token. Field values are always unsigned; a postfix exclamation point can be used to sign-extend them.

Architecture manuals have informal field specifications. For example, the fields for some SPARC load instructions are (SPARC 1992, p 90):

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i</th>
<th>simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29</td>
<td>25 24</td>
<td>19 18</td>
<td>14 13 12</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Other instructions may use a different format, e.g.,

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>opf</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29</td>
<td>25 24</td>
<td>19 18</td>
<td>14 13</td>
<td>5 4</td>
<td>0</td>
</tr>
</tbody>
</table>

for floating-point arithmetic. This fields declaration defines the fields used in these and all other SPARC instructions:

```plaintext
fields of itoken (32)
op 30:31 rd 25:29 op3 19:24 rs1 14:18
i 13:13 simm13 0:12 opf 5:13 rs2 0:4
op2 22:24 imm22 0:21 a 29:29 cond 25:28
disp22 0:21 aui 5:12 disp30 0:29
```
The first two indented lines define the fields used in the formats pictured above; the last two lines define fields used in SPARC formats that aren't pictured in this paper. Because all SPARC instructions are 32 bits wide, only one class of tokens is needed, the 32-bit itoken (mnemonic for "instruction token"). When instructions vary in size, more classes 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 1993, page 26-3):

\[
\begin{array}{|c|c|c|}
\hline
\text{ModR/M} & \text{reg/opcode} & \text{r/m} \\
\hline
\text{7} & \text{6} & \text{5} & \text{3} & \text{2} & \text{0} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{SIB} & \text{index} & \text{base} \\
\hline
\text{7} & \text{6} & \text{5} & \text{3} & \text{2} & \text{0} \\
\hline
\end{array}
\]

The fields declarations for these bytes are:

- fields of ModR/M (8) mod 6:7 reg_opcode 3:5 r_m 0:2
- fields of SIB (8) as 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.

2.2 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 range of values a field may have. The typical range has a single value, e.g., \( \text{op} = 1 \). Patterns may be composed by conjunction (\&), concatenation (;), or disjunction (|).

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. The constraint "\( 10 <= f < 81 \)" on a field \( f \) is tested against a single token of the class to which \( f \) belongs. The constraint matches that token if the \( f \) field of the token falls in the range defined by 10 and 81. The wild-card constraint "some class" matches any token of class class, for example, on the SPARC, "some itoken" matches any 32-bit token.
A conjunction "p & q" matches if both p and q match. We typically use conjunction to constraint multiple fields within a single token. A concatenation "p; q" matches if p matches an initial sequence of tokens and 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 somehow related, e.g., to group the SPARC integer-arithmetic instructions.

Finally, patterns can be labelled. The pattern L: p matches whenever p matches, and it binds the identifier L to the location in the instruction stream where p matches.

Section 4.1 describes the toolkit's representation of patterns and defines the meanings of the pattern operators in terms of their effects on that representation.

Patterns in Specifications

The patterns declaration binds names to patterns. Pattern bindings are typically used to define opcodes and to group related opcodes. For example, the name call is bound to the pattern that corresponds to the SPARC opcode call by

```plaintext
patterns call is op = 1
```

The pattern op = 1 matches any 32-bit token in which bit 31 is zero and bit 30 is one. Opcodes can be defined by multiple constraints, for example

```plaintext
patterns add is op = 2 & op3 = 0
```

Defining opcodes individually would be tedious, and the result would be hard to compare with the architecture manual, which uses opcode tables. The patterns declaration can bind a list of names if a generating expression appears on the right. Generating expressions are modeled on expressions in the Icon programming language, which can produce more than one value (Griswold and Griswold 1990). A generating expression is a pattern in which some integers have been replaced by expressions in brackets like {0 to 3}, which denotes the sequence of integers (0,1,2,3). These expressions are activated in left-to-right LIFO order, resulting in a list of patterns, each of which is bound to the corresponding name on the left. For example, the following declaration describes the first opcode table in the SPARC manual (SPARC 1992, p 227):

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

This definition binds the names TABLE_F2, call, TABLE_F3, and TABLE_F4 to the patterns op = 0, op = 1, op = 2, and op = 3, respectively. These names can now be used in the definitions of new patterns.

---

1 Conjunction is permitted if and only if the constraints that are conjoined refer to fields in tokens of the same class; this restriction enforces the rule against mixing fields from different classes of tokens. For example, on the Pentium, the pattern mod = 0 & index = 5 is permitted, but the pattern mod = 0 & index = 2 is not.
Most manuals give tables in which not every opcode is used. Unused opcodes can be bound to the special name "\_\_", which is permitted on the left side of a binding. For example, Table F-3 from the SPARC manual defines many of the arithmetic opcodes (SPARC 1992, p 228):

```
TABLE_F3

patterns = { add addcc taddcc wrxxx
             and andcc tsubcc wpsr
             or orcc taddccw urwm
             xor xorcc tsubctw wrtbw
             sub subcc mulcc fpop1
             andn andacc ell fpop2
             orn ornc rcl cpop1
             xnor xnorcc era cpop2
             addx addxcc rdxx jmpl
             _ _ _ rdprr rett
             umul umulcc rdwm ticc
             smul smulcc rttbr flush
             subu subu cc _ save
             _ _ _ _ restore
             udiv udiv cc _ _
             sdiv sdiv cc _ _ }

is

TABLE_F3 & op3 = { 0 to 63 columns 4 }
```

The expression {0 to 63 columns 4} generates the integers from 0 to 63 in the sequence (0, 16, 32, 48, 1, 17, 33, ..., 63), not the sequence (0, 1, 2, ..., 63), so that, for example, the name addcc is bound to the pattern op = 2 & op3 = 16. This trick makes it possible to use tables in which opcodes are numbered vertically.

### 2.3 Constructors

A constructor connects the symbolic and binary representations of an instruction by mapping a list of operands to a pattern. The toolkit's generator creates an encoding procedure for each constructor, so application writers can use constructors. Constructors can also be used within specifications; applying a constructor to a list of operands produces a pattern. Using constructors and patterns in each others' definitions helps a specification writer organize the description of a machine's instruction set.

Because assembly language is the most familiar symbolic representation of instructions, we designed constructor specifications so their left-hand sides resemble descriptions of assembly-language syntax: a constructor name and a list of operands. Operands may be separated by spaces, commas, brackets, or other punctuation. The punctuation has no effect on the binary encoding procedures generated by the toolkit, but it is used to generate encoding procedures that emit assembly language, and also to generate a grammar that recognizes assembly language.
The right-hand side of a constructor specification contains a pattern that describes the binary representation of the instruction specified. That pattern may contain free identifiers, which refer to the constructor's operands; such operands may be integers, or they may be patterns produced by constructors of a given type. For example, the following constructor describes the SPARC floating-point negate instruction:

```
constructors
  fnegs n, m is fnegs & rs2 = n & rd = m
```

This definition of the constructor `fnegs` relies on a previous definition of the pattern `fnegs`, which appears on the right-hand side; that definition is

```
patterns fnega is fpopl & opf = 0x5
```

Using the name `fnega` to refer both to a pattern and to a constructor may be confusing, but it is also desirable; architecture manuals normally use the same names in opcode tables and instruction descriptions. The toolkit's specification language makes the reuse possible by putting constructor names in a separate name space.

The specification of the constructor `fnegs` is not bad, but it is awkward to introduce integer operands `n` and `m` to refer to registers `rs2` and `rd`. We simplify by using field operands instead of integer operands.

```
constructors
  fnegs rs2, rd is fnegs & ra2 & rd
```

On the right-hand side, the identifier `rs2` stands for the pattern constraining the field `rs2` to be equal to the first operand. This specification has fewer names to keep track of, but it has a new shortcoming: the same names appear in the same order on both sides of `is`, using only slightly different notation. This conjunction of all operands with the opcode is common in RISC machines, so we provide a special abbreviation for it, in which the right-hand side is omitted:

```
constructors
  fnegs rs2, rd
```

This specification looks almost exactly like the suggested assembly-language syntax in the SPARC manual (SPARC 1992, p 144), but it has a precise semantics. The generated encoding procedure, which has the C declaration

```
void fnegs(unsigned rs2, unsigned rd);
```

has the side effect of emitting an `fnegs` instruction into the current instruction stream.

Not all operands are simple integers or fields. For example, the SPARC integer-arithmetic instructions take a second operand that may be a register or an immediate operand, depending on the value of the `i` field (SPARC 1992, p 84). Such an operand is properly represented by a pattern. To prevent users from supplying patterns that don't make sense in the context of such an operand, the specification writer must designate a type `T` to be associated with that operand and with one or more constructors. The only valid operands of type `T` are those produced by applying constructors of type `T`. For example,
our SPARC specification defines the constructor type *reg_or_imm* for the special integer-arithmetic operands. Since the operand can be a register or an immediate value, the specification defines register-mode and immediate-mode constructors:

**constructors**

\[
\begin{align*}
\text{rmode } rs2 & : \text{reg_or_imm is } i \equiv 0 \& rs2 \\
\text{imode } \text{simm13}! & : \text{reg_or_imm is } i \equiv 1 \& \text{simm13}
\end{align*}
\]

where \text{simm13}! denotes the field \text{simm13} interpreted as a signed integer. The identifier *reg_or_imm* is used as an operand in the definitions of the arithmetic constructors, for example

**constructors**

\[
\begin{align*}
\text{add } rs1, \text{reg_or_imm}, \text{rd}
\end{align*}
\]

Encoding procedures corresponding to ordinary, untyped constructors emit tokens into the current instruction streams, but encoding procedures corresponding to typed constructors have no side effects; they simply return values. The encoding procedures generated from *rmode*, *imode* and *add* have these declarations:

\[
\begin{align*}
\text{reg_or_imm}_\text{Instance} \text{rmode(} \text{unsigned } rs2); \\
\text{reg_or_imm}_\text{Instance} \text{imode(} \text{int } \text{simm13}); \\
\text{void add(} \text{unsigned } rs1, \text{reg_or_imm}_\text{Instance reg_or_imm}, \text{unsigned } rd);
\end{align*}
\]

**Specifying similar constructors**

Some architecture manuals describe instructions in alphabetical order; others group instructions with related syntax or semantics. The toolkit uses disjunction to define patterns that match any of a group of related instructions. These patterns can also be used to specify constructors. For example, the specification

**patterns arith**

\[
\begin{align*}
\text{is add | addcc | addx | addxcc | taddcc} \\
\text{sub | subcc | subx | subxcc | tsbuc} \\
\text{wmul | smul | wmulcc | emulcc | malecc} \\
\text{udiv | sdiv | udivcc | sdivcc} \\
\text{save | restore | taddctv | tsbctv}
\end{align*}
\]

**constructors**

\[
\begin{align*}
\text{arith } rs1, \text{reg_or_imm}, \text{rd}
\end{align*}
\]

avoids repeated specifications for the constructors *add*, *addcc*, *addx*, and so on. When the constructor name on the left-hand side denotes a pattern, each disjunct of the pattern is used to generate a constructor. The patterns declaration attaches a name to each disjunct so that the constructor name can be computed.

More generally, specification writers can form compound constructor names by joining patterns, strings, and fields using the `\*` symbol. On well-factored architectures, one can use compound names to write very concise specifications.
For example, the following fragment of the MIPS specification defines 16 op-codes, 3 floating-point formats, and 48 floating-point compare instructions:

```plaintext
patterns
c.cond is c.f | c.eq | c.ueq | c.olt | c.ule | c.sf | c.ngle | c.seq | c.ngl | c.lt | c.ngt
fieldinfo
   format is [ sparse [ s = 0, d = 1, w = 4 ] ]
constructors
c.cond"n","format fs, ft
```

The `fieldinfo` directive tells the toolkit that the `format` field has only three interesting values, 0, 1, and 4, and it gives mnemonic names to those values. When the toolkit enumerates the constructor definition, it loops over the disjuncts of pattern `c.cond` and over the interesting values of field `format`. On the right-hand sides, `c.cond` is bound to a disjunct, and `format` is bound to a constraint. With its implicit right-hand side, the single constructor definition is equivalent to a series of 48 definitions:

```plaintext
constructors
   "c.f.s" n, m is c.f & format = 0 & fs = n & ft = m
   "c.f.d" n, m is c.f & format = 1 & fs = n & ft = m
   ... "c.ngt.w" n, m is c.ngt & format = 4 & fs = n & ft = m
```

**Equations**

Some instructions have integer operands that cannot be used directly as field values. The most common are PC-relative branches, in which the operand is the target address, but the corresponding field contains the difference between the target address and the program counter. Constructor specifications may include equations that express relationships between operands and fields. Equations relate sums of terms with integer coefficients. Terms include operands, fields, and free variables, and they can be bit-sliced or sign-extended. For example, we can specify the SPARC branch instructions by relating the target address to the program counter and the displacement field:

```plaintext
constructors
   branch_a addr { addr = L + 4 * disp22! } is L: branch & disp22 & a
```

The equation in braces shows the relationship between `addr`, the target of the branch, `L`, the location in memory of the instruction, and `disp22!`, the sign-extended displacement field. The toolkit contains a simple equation solver used by both the generator and the translator. For encoding, the generator uses operands as inputs and computes the values of fields. For decoding, the translator uses fields as inputs and computes the values of operands. The solver discovers conditions that must be satisfied if the equations are to have a solution. For example, `branch(addr)` is well-defined only if `(addr - L) mod 4 = 0`. The encoding procedures emitted by the generator enforce such conditions, and decoding code matches constructor applications only when the associated conditions are satisfied. The solver is described more fully elsewhere (Ramsey 1995b).
Equations may use inequalities as well as equalities. The solver does not use the inequalities to help solve the equations, but it does generate code to check that any solution satisfies the inequalities, so they can be used to express constraints.

**Synthetic instructions and conditional assembly**

Defining patterns by applying constructors is most useful when defining "synthetic" instructions, i.e., instructions that are available in assembly language even though they are not part of the real machine. For example, the synthetic instructions bset (bit set) and dec (decrement) are defined in terms of the real instructions or and sub (SPARC 1992, p 86):

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

`imode` converts the integer operand `val` into an operand of type `reg_or_imm` for use by `sub`.

Sometimes the best expansion for a synthetic instruction depends on the values of operands. We can choose one of several expansions by putting alternatives on the right-hand side of a constructor specification, each with its own set of equations. Each application of the constructor uses the first alternative for which the equations can be solved. In the example below, we specify the SPARC synthetic instruction set, which has three ways to load a signed value `val` into register `rd`. When the 10 least significant bits of `val` are zero, it uses a single `sethi` instruction to set `rd` to the 22 high bits of `val`. When `val` fits in 13 bits, it uses an immediate-mode or instruction where the first operand is register 0, which is always zero. Otherwise, it uses two instructions: `sethi` to assign the high-order bits and `or` to add the low-order bits:

```plaintext
constructors
    sethi val, rd is sethi & rd & imm22 = val[10:31]
    set val, rd
    when { val[0:9] = 0 } is sethi(val, rd)
    when { val = val[0:12] } is or(0, imode(val), rd)
    otherwise is sethi(val, rd); or(rd, imode(val[0:9]), rd)
```

These definitions use bit-slicing; for example, `val[10:31]` denotes the most significant 22 bits of the 32-bit integer value `val`. Bit-slicing can also be used to split operands among two or more fields, as with the `sh` operands used in the `XS` form of instructions on the PowerPC architecture\(^2\) (May et al. 1994).

---

\(^2\)The PowerPC documentation refers to such operands as “split fields.”
```plaintext
<table>
<thead>
<tr>
<th>Constructors</th>
<th>Eaddr</th>
<th>reg</th>
<th>mod</th>
<th>r_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg reg</td>
<td>Eaddr</td>
<td>reg</td>
<td>mod</td>
<td>r_m</td>
</tr>
<tr>
<td>Index [reg]</td>
<td>Eaddr {reg != 4, reg != 5}</td>
<td>i32</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Disp32 d[reg]</td>
<td>Eaddr {reg != 4}</td>
<td>i32</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Index [base][index * ss]</td>
<td>Eaddr {index != 4, base != 5}</td>
<td>index &amp; base</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Index8 d[base][index * ss]</td>
<td>Eaddr {index != 4}</td>
<td>index &amp; base &amp; ss; i8 = d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index32 d[base][index * ss]</td>
<td>Eaddr {index != 4}</td>
<td>index &amp; base &amp; ss; i32 = d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShortIndex d[index * ss]</td>
<td>Eaddr {index != 4}</td>
<td>index &amp; base &amp; ss; i32 = d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 1: Constructor definitions for the Pentium's 32-bit addressing modes

2.4 CISC instructions

All MIPS and SPARC instructions can be specified by conjoining field constraints; this is the property that makes it useful to specify constructors implicitly by omitting the right-hand sides. The Pentium is not so simple. Both opcode and operands can span several tokens, and some tokens contain parts of each. Fields have multiple uses; for example, the field r_m can indicate either a register choice or an alternate addressing mode, depending on its value. Figure 1 shows constructor specifications for the Pentium's addressing modes, illustrating how the toolkit's specification language handles the complexity of CISC. The brackets and asterisks do not affect encoding and decoding, but they are used to derive an assembly syntax for the constructors. The toolkit can generate "alternate encoding procedures" that emit this syntax instead of a binary representation, and it can generate a grammar that recognizes this syntax and calls the appropriate encoding procedures. The brackets and asterisks also serve as mnemonic devices; they show the relationship of the constructors to the Intel assembly language. Figure 2 shows the structures of the patterns used in Figure 1.

Effective addresses contain 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. Finally, some modes take immediate displacements (Intel 1993, Tables 26-2 to 26-4). None of the Pentium addressing modes specifies a value for the reg_opcode (middle) field of the ModR/M token. That's because 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. When an effective address is conjoined with a pattern constraining reg_opcode, the result is a complete specification of an instruction.
We define constructors of type Eaddr to create effective addresses in 32-bit mode. The first group of constructors specifies the non-indexed 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 eight general registers. The next three modes are register-indirect modes with no displacement, 8-bit displacement, and 32-bit displacement. The fields $mod$ and $r_m$ of the ModR/M token are defined above; the fields $i8$ and $i32$ occupy full 8-bit and 32-bit tokens and are used to hold displacements:

- fields of $i8$ (8) $i8$ 0:7
- fields of $i32$ (32) $i32$ 0:31

Semicolons separate ModR/M tokens from the displacement tokens that follow. The inequality $reg \neq 5$ 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 $reg \neq 4$ in the equations associated with the register-indirect modes 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 1 and 2. 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 ($index \neq 4$). 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.
2.5 Relocation

Instructions often refer to the addresses of data or of other instructions; e.g., to load the value of a variable or to branch to a label. The toolkit can emit such instructions even before the addresses are known. Instructions and data are emitted into relocatable blocks, which implement the instruction-stream abstraction. An application can write into a relocatable block without knowing where in memory the block's contents will eventually be located. Relocation assigns an address to a relocatable block. The contents of a relocatable block always occupy contiguous memory locations; the relocatable block is the smallest unit that can be moved independently (i.e., it is the unit of relocation). Applications may use any number of relocatable blocks. A simple UNIX assembler, for example, might use only three: one each for code, initialized data, and uninitialized data.

A label points to a location in a relocatable block. The toolkit does not associate names with labels; applications can use any method they want to find and identify labels. A relocatable address is any quantity whose value depends on the value of a label. The toolkit's generator treats relocatable addresses as values of an abstract data type. There are only two operations on the type. One asks whether the value of the address is known; the other, which may be used only when the value is known, produces the value, which is an integer.

The toolkit's library provides one implementation of relocatable addresses; to the library, a relocatable address is the sum of a label and a signed offset. This simple form is adequate for applications like compilers and linkers. Authors of more sophisticated applications can use more sophisticated representations (e.g., linear expressions over addresses and labels) without changing the toolkit. For example, some application writers might like relocatable addresses to include constants, labels, differences of two labels, and any sum of these elements.

As might be guessed from the abstraction, the toolkit's specification language does not limit relocation to those operands that truly denote addresses. Any integer-valued operand can be designated relocatable by giving its name after the relocatable keyword. For example, the following declaration is used to make the addr operand of the SPARC branch constructor relocatable:

relocatable addr

In addition to operands that are designated relocatable, all pattern labels that appear on right-hand sides of constructor definitions are relocatable.

Allowing any operand to be relocatable simplifies the adaptation of applications originally intended to work with assembly language. Most assembly languages permit some use of relocatable operands where integers are expected. For example, in the GNU assembly language for the Pentium, a d or a operand to an effective-address constructor like Abs32 and Index32 may be an integer, the name of a label, or a sum of the two. This flexibility allows applications that emit assembly code to write addressing expressions that involve labels, because

---

3This "label" is different from the labels introduced by the L:p construct, although both serve the same function. In sections where we discuss both kinds of labels, we refer to the L:p labels as "pattern labels."
the applications can rely on the assembler and linker to perform the address arithmetic and relocate the instructions. Relocatable operands give this flexibility to applications that emit binary, because the toolkit's representation of relocatable addresses supports a limited form of symbolic arithmetic on labels. This flexibility significantly simplifies the construction of mId's code generators, because it enables automatic translation of existing assembly-emitting code generators into mId's binary-emitting code generators. Without the ability to make any operand relocatable, large parts of mId'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 tokens and creates a relocation closure. The closure contains references to the unknown addresses, plus a pointer to a function that, when applied, overwrites the placeholder with the correct instruction. The application keeps the closure until the addresses it depends on become known, at which point it can apply the closure function and discard the closure.

For flexibility, we let applications decide how to organize relocation closures, when to apply them, and when to discard them. For example, a standard linker might store all closures in a simple list and discard them after applying them, because the absolute addresses of segments don't change after they are assigned. An incremental linker would keep the closures, because some might have to be re-applied when relocatable blocks were moved. It might store the closures in a more complex data structure, to avoid re-applying all closures when only a few relocatable blocks moved.

For placeholders to be computable, the specification writer must associate a placeholder pattern with each class of tokens. The toolkit uses the shape of a constructor's pattern to compute a placeholder for it, ensuring that the placeholder has the same shape as the instruction that overwrites it when the closure is applied. Placeholders are typically chosen so that attempts to execute them are detected. For example, we chose

\[
\text{placeholder for itoken is unimp & imm22 = Oxbad}
\]

as the placeholder for the SPARC. A dynamic linker might use a special trap instruction as a placeholder; it could handle the special traps by resolving the unknown address and applying the instruction's closure at run time.

When a conditionally assembled constructor is applied to a relocatable address, it may not be possible to determine which sets of equations can be satisfied, because the value of the relocatable address may not be known. In that case, the toolkit makes the most conservative decision, choosing the first alternative whose equations are known to be satisfied. This technique, while safe, is not suitable for emitting span-dependent instructions; for example, it uses the most general representation for all forward branches.
2.6 Matching statements

Decoding applications use the toolkit's matching statements. These resemble ordinary case statements, but their arms are labeled with patterns. The first arm whose pattern matches is executed. Free identifiers used in these patterns are binding instances; they are bound either to field values or to the locations of sub-patterns within the pattern. For example, the matching statement

```plaintext
match p to
| fnegs & rs2 = n & rd = m => printf("let f%d = - f%d", n, m);
| some itoken =>
endmatch
```

prints a message if the instruction pointed to by p is a floating-point negate, and it does nothing otherwise. The pattern `some itoken` always matches, so if the two arms of this matching statement were reversed, the toolkit's translator would issue a warning that the second arm could never be executed.

Just as in specifications, it is often more convenient to write patterns in the form of constructor applications, e.g., `fnegs(n, m)`, in which the operands `n` and `m` are bound to integers by the matching statement. Pattern-valued operands are bound not to integers but to locations in the instruction stream; for example, the SPARC pattern `add(0, operand2, rd)` matches any add instruction in which `rs1` is zero, and it binds `rd` to the destination register and `operand2` to the location of the token containing the second operand. Pattern labels are also bound to locations in the instruction stream. The generated decoder converts all such bound locations to integers, and those integers can be used on the right hand side of the arrow (=>).

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 applications 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. Widths are measured in bits; offsets are measured in the same units used for the program counter, which defaults to 8 bits per addressing unit. The application writer must supply code that can fetch tokens using the proper byte order, which is usually the byte order of the machine the application runs on. Section 3.1 shows token-fetching code and a nontrivial matching statement that are used in ldb.

3 Using Specifications and the Toolkit

This section shows how we use the toolkit to build applications. We begin with descriptions of two applications; an optimizing linker, which uses the toolkit to encode instructions, and a debugger, which uses the toolkit to decode instructions. Using the toolkit in applications forces resolution of several implementation issues. One must find names for toolkit-related code, and in the absence of
language support for overloading; they must be unique. We use two strategies for avoiding name clashes. The toolkit guarantees internal consistency of encoding and decoding, but it cannot guarantee consistency with the real machine; we use a specification checker to test specifications for consistency with independently developed assemblers and disassemblers. We complete the section by describing our implementation; the toolkit itself runs slowly, but it generates efficient code.

3.1 Using the toolkit in applications

Figures 3 and 4 show how the toolkit is used in two applications. Code is shown in boxes, data in ovals. Code in doubled boxes is machine-dependent; a version exists for each target architecture. Code in single boxes is machine-independent. Code pointed to by thick, dashed arrows is generated by the toolkit. Boxes with heavy borders contain code that is part of the toolkit or generated by the toolkit. Ovals with heavy borders contain instruction streams that are written or read by toolkit-generated code. The names of the three parts of the toolkit are shown in italics. The same machine descriptions are used for both applications.

mld

mld, shown in Figure 3, is a retargetable, optimizing linker for the MIPS, SPARC, and Intel 486. mld links a machine-independent intermediate code, optimizes it, generates instructions and data, and emits a machine-dependent executable file (a.out). Retargeting mld requires adapting a code generator and writing code to emit an a.out file.

mld uses the toolkit's encoding procedures, library, and relocation mechanism. Like other encoding applications, mld provides a routine that the library uses to allocate memory for closures, labels, and relocatable blocks. The library provides routines that create blocks and labels, select a block and location for emitting data, and emit integers. The generator creates the encoding procedures.

mld uses instruction streams to model segments in an executable file. For example, a SPARC executable contains instructions in a "text" segment, initialized data in a "data" segment, and uninitialized data in a "bss" segment. mld uses one instruction stream each for the data and bss segments. It divides the text segment into many instruction streams, one for each procedure. mld can place procedures in memory in different orders; different procedure placements cause significant variations in elapsed-time performance of programs generated with mld (Fernández 1995).

mld's code generators are based on those used in the 1cc compiler (Fraser and Hanson 1995), which emit assembly code. Most of the code for 1cc's code generators is generated automatically from 1burg specifications (Fraser, Hanson, and Proebsting 1992); these parts rewrite intermediate-code subtrees to assembly-language templates. Some of the code is written by hand; these parts handle target-specific tasks that are not amenable to automatic generation, like
adapt a code generator means modifying both the BURG specification and the hand-written parts.

Because the toolkit handles encoding, error checking, and relocation of instructions, much of the translation of lcc's assembly code generators to mld's binary code generators is trivial. The translation is automated by a script that matches assembly strings in the source of lcc's code generators and replaces them by calls to encoding procedures. For example, on the SPARC, lcc emits an instruction to allocate a stack frame by executing

```c
printf("save %s\n", -framesize);
```

mld calls a toolkit-generated encoding procedure:

```c
save(SP, imda(-framesize), SP);
```

where SP is #defined to be 14, since register 14 is used as the stack pointer. The strong correspondence between assembly syntax and the signatures of en-
coding procedures permits automatic translation of lcc's stable assembly code generators into mld's binary code generators, and that automation reduces the possibility of errors.

mld uses calls to the toolkit's library in lieu of assembler directives. For example, lcc uses

```c
printf("skip \%d\n", n);
```

to allocate uninitialized space in the current segment; mld uses

```c
addlc(n);
```

to advance the location counter of the current relocatable block, having the same effect.

While emitting instructions and data, mld saves each relocation closure on a list associated with the current relocatable block. After emitting all procedures, it assigns addresses to all blocks, then applies the saved closures, which do the relocation. Finally, it writes the a.out header into a file and calls library procedures to write the relocatable blocks into that file. Because the generated closure functions check for errors and replace placeholders with relocated instructions, mld needs only 20 lines of C code for relocation, and that code is machine-independent.

mld exposes a deficiency in the toolkit's support for decoding. During code generation, mld builds new addressing expressions from existing ones. For example, it might build an Index32 operand from an Index operand. The encoding procedures for these constructors, however, produce opaque values, which cannot be decoded without first emitting them into a stream. mld distinguishes
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); L: epsilon => RETURN FollowSet.T{L, target};
  | branch.a(target) & (ba | fba | cba) => RETURN FollowSet.T{target};
  | jmpl(dispA(ra1, simm13), rd) => RETURN FollowSet.T{GetReg(m, ra1)+simm13};
  | jmpl(indexA(rs1, rs2), rd) => RETURN FollowSet.T{GetReg(m, rs1)+GetReg(m, rs2)};
  | some itoken => Error.Fail("unrecognized instruction");
endmatch
END Follow;

Figure 5: Matching statement used for control-flow analysis of SPARC instructions

such values by breaking the typed-constructor abstraction and peering at their representation. This solution is unsatisfying; it violates data abstraction, and it is likely to be incompatible with changes that are planned to improve the toolkit's performance. The toolkit should provide an official mechanism for matching on intermediate values produced by constructor applications.

ldb

ldb, shown in Figure 4, is a retargetable debugger for ANSI C. Most of its breakpoint implementation is machine-independent; the only machine-dependent part is the analysis of control flow (Ramsey 1994a). The analysis is written using a matching statement. ldb also uses matching statements to print assembly-language representations of instructions.

ldb is written in Modula-3. It uses an object type to represent an instruction stream of a program being debugged, and it uses unsigned integers to refer to locations in such streams. Here are the code fragments that give the toolkit's translator the representation of streams:

address type ia "Word.T"
address add using "Word.Plus(%a, %o)"
address to integer using "%a"
fetch any using "FetchAbs(m, %a, Type.I:d).n"

The quoted strings are fragments of Modula-3 code in which %a stands for an address or location, %o stands for an offset, and %w stands for a width. Offsets and widths are measured in bits. Word.Plus is an unsigned add. The m argument to FetchAbs is an object representing the address space being debugged; it must be defined by the context in which matching statements appear. The decoders generated by the toolkit use these fragments to manipulate instructions.

Matching statements make flow analysis clear and concise. Figure 5 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 jmp1 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 5 expands to nested case statements totaling about 90 lines of Modula-3 code. The count does not really convey the difficulty of writing the code by hand; the toolkit combines seemingly unrelated opcodes if they result in executing the same code. Using the matching statement implemented by the toolkit makes it clear what the code is doing; the logic would be obscured if implemented by nested case statements.

3.2 Naming issues

The names of instructions may conflict with names that application writers use; generating encoding procedures with those names can cause name-space collisions. Different languages provide different mechanisms for solving collision problems; for example, C++ has classes and Modula-3 has interfaces. C uses a single name space, but one can attach a unique prefix to the names of encoding procedures, or one can refer to them indirectly by using a structure containing function pointers. We use the second alternative, because it enables a single application to use multiple sets of encoding procedures, e.g., one to emit binary and one to emit ASCII. We use sets of encoding procedures in our specification checker, which is described below.

A more difficult problem is that the names of instructions may conflict with each other. Many assemblers overload instruction names, using context to determine which instruction is meant. For example, in Pentium assembly code, the add opcode can represent any of five different instructions, depending on the sizes and locations of its operands. Because not all target programming languages support overloading, the toolkit must use different names for different instructions (constructors), lest the names of encoding procedures collide. Even in languages that support overloading, their name-resolution mechanisms may be less powerful than an assembler's parser. This naming problem is not limited to machine descriptions; anyone providing an API for a language-based service may face it in another guise.

We require that each constructor have a different name. A typical specification distinguishes variants using suffixes. For example, our Pentium specification includes constructors called addb, addib, addiw_ob, addbr, and addbrm. Distinguishing variants like these is necessary, but it may make specifications harder to understand, since architecture manuals often use the shorter, overloaded names. The distinction complicates the adaptation of applications that
use (overloaded) assembly language, because toolkit-based applications must use the full names. In adapting the lcc code generators to make aI4, we observed cases that were treated as a single "instruction" in assembly language but that had to be split into multiple cases when using the toolkit.

Overloading also makes it harder to generate assemblers or the assembly-emitting procedures used to help check specifications. We solve this problem by defining mappings from full names to names used in assembly language, as described in Section 5.1. These mappings are separated from the main machine description, because different vendors use different syntaxes for their assembly languages.

3.3 Specification checkers

The toolkit forbids erroneous and internally inconsistent specifications. Furthermore, it warns of implausibilities in specifications. An implausible constructor specification is technically legal but has some property that suggests it might be in error; e.g., the constructor is untyped but does not determine the values of all the bits in the tokens it constrains. Checking for implausibility can catch many mistakes in a specification, but it cannot help when a specification, while remaining plausible, does not accurately describe the target machine. For example, our original Pentium specification was plausible, but it contained many errors. We mistranscribed some opcode definitions; for example, we initialized some fields to incorrect values, and we transposed names in opcode tables. We also used the wrong operands in places; for example, we specified an unsigned operand where a signed operand was required, and we used a 32-bit operand where an 8-bit operand was required. We discovered all these errors by automatically checking our specification against the GNU assembler for the Pentium.

A toolkit specification provides a bidirectional mapping between symbolic and binary forms of instructions. If we express the symbolic form in a suitable assembly language, we can check the mapping for consistency by composing the toolkit's map with the inverse map provided by an independent assembler or disassembler. Figure 6 shows the desired property diagrammatically: all paths

![Figure 6: Equivalence of assembly and binary representations](image-url)
from an instruction to an assembly or binary representation should produce identical representations of that instruction. To enable the left downward arrow in Figure 6, the toolkit can generate procedures that have the same declarations as standard encoding procedures but that print assembly language instead of emitting binary. We discuss the specification of the assembly language in Section 5.1. Given this facility, we could, in principle, check a specification as follows:

1. Choose a sequence \( I \) of instructions to test. The sequence should provide "good coverage" of the specification; for example, it should exercise every constructor at least once.

2. Use the toolkit to encode sequence \( I \) in an assembly file \( A \in A \) and a binary file \( B \in B \).

3. Apply the independent assembler to \( A \), producing a binary file \( B' \), and similarly disassemble \( B \) to produce \( A' \).

4. If \( A \) differs from \( A' \) or \( B \) from \( B' \), there is an inconsistency among the assembler, disassembler, and toolkit specification.

The choice of instructions to test is independent of the other steps, and we present it first.

Deciding what instructions to test

It is impractical to check every encoding of every instruction; there are too many encodings overall. Variations in operands' values produce most of the encodings, so it is practical to check every constructor with at least a few operand values. Our specification checker must be told which operand values to try. For example, if the input to the checker contains the phrase r32 [ 2 7 ], then the checker uses the values 2 and 7 wherever an operand named r32 is called for in the specification. The checker builds a table associating the names of operands with sets of values. It then enumerates all the untyped constructors in the specification. For each constructor, it enumerates all the values of the operands associated with that constructor, and it emits assembly and binary forms by applying the constructor to each combination. If an operand is of a constructor type, not a field or integer type, the checker repeats the enumeration process recursively on constructors of that type. This recursion explores the entire state space defined by the constructors and the given sets of operand values. The state space can be adjusted by associating different sets of values with an operand name, depending on the type of the constructor in which the operand is to be used.

We can imagine several improvements to the checker. One would be to generate the sets of operand values automatically from the machine description. Interesting sets might include boundary values, i.e., the limits of ranges, as well as such perennial troublemakers as 0, 1, and -1. One might include a variable
number of values chosen at random, depending on how much time one is willing to spend checking for errors.

Another improvement would be to refrain from using suggested values when such use would violate the conditions associated with a constructor. For example, 5 is normally an acceptable value of a Pentium operand named \texttt{reg}, but not when the constructor in question is \texttt{Indir}, because that constructor includes the constraint \texttt{reg \neq 5}. It is tedious to keep problematic values out of value sets; we would prefer that the checker detect and avoid such values at constructor-application time.

A third improvement would be to search the state space non-exhaustively. Hashing the state space into a bit vector of a size specified at checking time would enable partial exploration of a state space much larger than could be explored exhaustively. A similar technique is used in the SPIN protocol validator (Holzmann 1988).

Testing the instructions

There are a few problems with the simple picture of testing shown in Figure 6. First, it assumes that assembly and disassembly are exact inverses, whereas in practice they are usually only approximate inverses. If we denote these transformations by \( \alpha : A \rightarrow B \) and \( \delta : B \rightarrow A \), then \( \alpha \circ \delta \) may be the identity function on \( B \), but \( \delta \circ \alpha \) is seldom the identity function on \( A \), because assembly language usually has more than one way to represent a single binary instruction. A more formal way to state this problem is that the range of \( \delta \) is only a subset of \( A \) and that the restriction of \( \alpha \) to that subset is an inverse of \( \delta \). Sometimes the output of the disassembler \( \delta \) is not in \( A \) at all, i.e., it is not valid assembly language. In that case \( \alpha \) and \( \delta \) are neither right nor left inverses and we can only compare in \( B \). Comparing in \( B \) is sufficient to establish the presence or absence of an inconsistency, but it is much more useful to compare in \( A \), because \( A \) (assembly language) is designed to read by human beings, and differences between \( A \) and \( A' \) may not only establish the presence of an error, but may also show where in the specification the error lies and possibly even what the error is.

A final problem is that \( B \), which is \( \alpha \)'s range and usually also \( \delta \)'s domain, is not a simple sequence of instructions but rather an object file, in which the instructions are surrounded by headers and other information, the format of which depends on the machine and operating system involved. It is not too much work to write code that emits stripped-down object-file headers, but there is a better way.

Our checking technique exploits the many-to-one nature of the assembly mapping \( \alpha \), and in particular the fact that every token in a binary file in \( B \) has a natural representation in \( A \) that is independent of the toolkit specification. That representation is whatever pseudo-operations the assembler uses to place data into an instruction stream. For example, the Pentium instruction \texttt{addb $127,%a1} is represented by two 8-bit tokens with values 4 and 127, and the GNU assembler can be told to emit those tokens by the pseudo-operations

\begin{verbatim}
.byte 0x4
.byte 0x7f
\end{verbatim}
Let us call these pseudo-ops \( A_{\text{data}} \), while we call the more usual representation \( A_{\text{instructions}} \). We can provide the toolkit with "binary" emitters that emit pseudo-ops in \( A_{\text{data}} \). We can then compare the two versions by assembling both representations into \( B \), and we can facilitate the comparison by disassembling back into \( A_{\text{instructions}} \), as shown in Figure 7. If the disassembler maps into some other language \( \overline{A} \), which is not a valid input to the assembler, it doesn't matter; the comparison in \( \overline{A} \) is used only to help the human being, so any \( \overline{A} \) that is human-readable works.

We combine the two branches of Figure 7 by generating another set of encoding procedures, which emit into a single stream of assembly language first the "instruction" and then the "data" version of each instruction in \( I \). This assembly-language output is then assembled and disassembled. If, as is usual, the disassembler writes one line of output for each instruction, inconsistencies can be identified by scanning the output with a Unix shell script which extracts pairs of lines that differ.

An aggressive assembler can create false positives, i.e., differences in the assembly and binary representations that aren't caused by inconsistencies between the toolkit specification and the actual machine. For example, on the Pentium, the ShortIndex effective address 4[\%eax+1] is more compactly encoded by the Diap32 effective address 4[\%eax], and when the longer ShortIndex form is expressed in \( A_{\text{instructions}} \), the assembler rewrites it into the shorter Diap32 form before encoding it. The toolkit, by contrast, does exactly as it is told and encodes the longer version into \( A_{\text{data}} \), and the result is a "false" mismatch after both the \( A_{\text{instructions}} \) and \( A_{\text{data}} \) versions are assembled into \( B \). This kind of false positive appears in both the intermediate \( B \) and the final \( A \) versions. Luckily, there are not many of these constructs and the code that generates \( I \) can be rigged to avoid them.
### 3.4 Implementation

Prototype versions of the toolkit’s generator and translator are 6000 lines of grammar, rewrite rules, and Icon code (Griswold and Griswold 1990). The library is 800 lines of ANSI C. The specification checker is a separate Icon program forked from an earlier version of the generator. Table 1 shows some characteristics of our three machine specifications, including the time the prototype generator takes to produce complete sets of encoding procedures. The number of addressing modes affects that time, because each encoding procedure has an alternative for each mode of each operand. The long time required to generate Pentium procedures is unsatisfying, but it remains workable because one rarely writes a specification containing hundreds of new constructors. One can add a few constructors to an existing specification in time proportional to the number of added constructors, not to the size of the whole specification.

The translator takes 10 seconds to transform either idb’s SPARC follow-set matching statement (Figure 5) or the analogous MIPS statement into Modula-3 code. The translator time, although shorter than the generator times, is more problematic, because the translator must be run after every change to a source file with a matching statement.

The toolkit generates efficient code. mid, our example encoding application, can use the toolkit to emit binary, or it can emit assembly code. It always executes faster when emitting binary. For example, when linking and emitting binary code for the integer SPEC benchmarks, mid is up to 15% faster than when it emits assembly code, as shown in Table 2. Moreover, emitting assembly requires running the assembler, which increases the total time required to generate an a.out without using the toolkit: 1.7–2.1 times longer on the SPARC and 2.8–5.6 times longer on the MIPS. This comparison is unfair to the MIPS assembler, because the MIPS assembler schedules instructions but the toolkit does not.

Application writers can trade safety for more efficiency. By default, the toolkit checks the widths of field values, calling a user-defined error procedure.
Table 2: Seconds to generate code & make a.out

<table>
<thead>
<tr>
<th>Program</th>
<th>MIPS</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bin</td>
<td>Asm, Run</td>
</tr>
<tr>
<td>eqlott</td>
<td>4.1</td>
<td>4.3 + 7.1</td>
</tr>
<tr>
<td>li</td>
<td>7.2</td>
<td>7.5 + 14.7</td>
</tr>
<tr>
<td>espresso</td>
<td>14.7</td>
<td>17.5 + 33.0</td>
</tr>
<tr>
<td>gcc</td>
<td>52.3</td>
<td>60.5 + 233.2</td>
</tr>
</tbody>
</table>

Bin out: Use toolkit to emit binary a.out
Asm out: Emit assembly code, without toolkit
Run as: Translate assembly code to a.out

if they overflow. Application writers unwilling to pay for a compare and branch can direct that field values silently be narrowed to fit. Those unwilling to pay even the cost of masking out high-order bits can assert that certain fields never overflow, in which case the values are used without masking. This choice is appropriate in some situations, for example, when field values denote registers and are chosen by a register allocator.

We measured encoding costs on a DEC 5000/240 with a memory-mapped clock. Simple encoding procedures like nop and mov cost less than 30 cycles when generated without safety checks; 6 of these cycles are for procedure call and return. Safety checks add 2 cycles per operand checked. Encoding a branch instruction, which requires checking relocatable addresses and doing a relative-address computation, costs 118 cycles.

The prototype generator and translator implement the complete specification language described in this paper. They are "prototypes" because they are slow, and because they stop processing specifications after the first error. We expect to improve their speed and usability by rewriting them in Standard ML (Milner, Tofte, and Harper 1990).
Combine with to make Matching rule is

- disjuncts | disjunction (pattern) match any disjunct
- sequents ; sequence (disjunct) each sequent matches a token
- constraints & conjunction (sequent) token satisfies every constraint
  range constraint field falls within range

Figure 8: Components of the normal form of patterns

4 Creating Encoders and Decoders

The core of the toolkit’s intellectual contribution is its specification language, together with the representations and algorithms used to turn specifications into encoders and decoders. Most of the algorithms follow from our choices of representations for patterns and constructors, which we develop and explain here, in two stages. The representations alone don’t obviously lead to efficient decoders, so we also describe our heuristic algorithm for finding efficient decision logic to use in decoders.

4.1 Representing patterns and constructors

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 8 shows the components of the normal form. The terminology can be confusing, because we may use any of 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, or list of alternatives. An empty list is permitted; the empty disjunction never matches. 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, which is denoted by \texttt{epsilon}, always matches, consuming no tokens. Finally, a conjunction matches a token if the token satisfies all of the range 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 \texttt{some class}, is permitted; it matches any token of the associated class.

We use the association between conjunctions and token classes to define the shape of a sequence, which is the list of associated classes. 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.

\footnote{One can obtain an empty disjunction by, for example, conjoining two mutually exclusive constraints.}
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 wild-card constraint has a form in which the sequent contains no constraints, i.e., it is the empty conjunction.) The normal forms of \( p \lor q \) and \( p; q \) are straightforward. We form \( p \lor q \) 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 \( p \land q \) by distributing over disjunction, but the rules for conjoining two sequences are more complicated. The two sequences must have the same shape, i.e., they must be the same length, and the associated classes of the corresponding sequents must be the same. We conjoin two sequences of identical shape by conjoining their individual sequents, elementwise. Conjoining two sequents simply means conjoining their constraints. It is not hard to show that these mappings to normal form, combined with the rules for matching in normal form, imply the matching properties given in Section 2.2.

Using only contiguous ranges in field constraints simplifies many parts of the toolkit's implementation, but a defect of this approach is that one cannot represent an inequality like \( \text{reg} \neq 4 \) as a field constraint. The "field constraint" \( \text{reg} \neq 4 \) is syntactic sugar for \( \text{reg} < 4 \lor \text{reg} > 4 \). This expansion is adequate for use in decoding, but not in encoding. We could change the definition to make a field constraint force a field to lie in a finite union of intervals, but it is not clear that the gain in expressive power would justify the extra complexity in the generator and translator. We have managed by using inequalities in equations, as shown in Figure 1.

Liberalizing conjunction

Under the rules we have given for conjunction, there is no pattern that can be conjoined with all of the effective addresses used on the Pentium. Figure 2 shows that the different effective-address constructors produce patterns of different shapes. If we can't conjoin these effective addresses with a pattern like reg-opcode = 0, we can't ever specify a value for reg-opcode. The shape of reg-opcode = 0 is ModRM, so it can be conjoined with the Reg or Indir forms of effective address, but not, for example, with the Disp8 form, whose shape is ModRM; 18. The solution is to liberalize our rules for conjoining sequences. If the shape of one sequence is a prefix of the shape of another, we can conjoin the 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 trick works when one sequence is a suffix of another.

If the toolkit used these prefix or suffix tricks automatically, it might silently accept a conjunction that the specifier didn't intend, so it uses them only when the specification tells it to. 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

\[5\] As distinguished from the inequality \( \text{reg} \neq 4 \), which appears only in equations.
conjoined with the prefix of \( p \) whose shape matches its shape, and the rest of \( p \) is concatenated to the result. Similarly, \( \ldots \quad \text{q} \) 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 also has the effect of making a pattern "lose its shape" where the ellipsis appears, so \( \ldots \quad \& \quad \ldots \quad \text{q} \) is never legal, because \( \ldots \) has no well-defined suffix and \( \ldots \quad \text{q} \) has no well-defined prefix. To represent the ellipsis, we change the normal form by adding a Boolean flag to each end of each sequence, showing whether the ellipsis is present at that end of the sequence.

Using the ellipsis makes it possible to constrain the \text{reg\_opcode} part of a Pentium effective address "after the fact." Because all the effective addresses have shapes beginning with ModRM, it is legal to write \text{Eaddr} \quad \& \quad p \ldots \) whenever \( p \)'s shape is ModRM. For example, \( \text{Eaddr} \quad \& \quad \text{reg\_opcode} = 0 \ldots \) is a legal pattern.

**Introducing free variables**

The representation of patterns described above cannot accommodate patterns with free variables. Free variables may appear in patterns in two contexts. First, constructors are defined by patterns with free variables. In this case, all the free variables must be operands of the constructor being defined, that is, they must be bound by the constructor's definition. The second context is in matching statements, where 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 on the right-hand side of the arm labelled by the pattern.

Having free variables in patterns complicates their representation. When a range constraint appears in a sequent, it constrains a field to fall in a range that is known statically. We add a new specification element, the field binding; when it appears in a sequent, it constrains a field to be equal to a value computed dynamically. The dynamic computation is represented as an expression containing free variables. The syntax of a field binding is the same as that of a range constraint, namely \( f = \text{expression} \). If the expression evaluates to an integer constant, this syntax denotes a range constraint; otherwise it denotes a field binding.

Using a field binding assumes that the value of the expression fits into the field. We make that assumption explicit in the normal form for patterns by generating a test to verify that the value falls into a suitable range. This test becomes a \textit{condition} of matching, and we associate such conditions with each disjunct. Although the conditions could be associated with each range 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. Such conditions may be written explicitly in specifications, or they may be consequences of equations. 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.
Figure 9: Extended normal form of patterns, with matching and encoding rules

Figure 9 shows the real representation of patterns, together with the rules for matching and encoding them. Disjuncts also have two Booleans that indicate the presence or absence of the ellipsis (\ldots) at the left and right ends of the sequence, but these Booleans are omitted from Figure 9 because they play no role in matching or encoding. The next section describes the role that field bindings, conditions, and equations play in matching.

Representing and encoding constructors

Adding field bindings and conditions to the representation of patterns makes it possible to represent constructors as lambda terms of the form \(\lambda x_1, \ldots, x_n.p\) where \(x_1, \ldots, x_n\) are the constructor's operands and \(p\) is a pattern with free variables \(x_1, x_2, \ldots, x_n\).

For each untyped constructor, the toolkit emits an encoding procedure with formal parameters \(x_1, x_2, \ldots, x_n\), the body of which encodes \(p\). For each typed constructor, the toolkit emits an encoding procedure that returns a structure identifying the constructor and containing its operands. The patterns corresponding to typed constructors are “inlined” in the untyped constructors where they are used.

The rules for encoding a pattern \(p\) are shown in Figure 9. The typical pattern \(p\) has exactly one disjunct (sequence), each sequent of which corresponds to a token to be emitted into the instruction stream. The constraints and field bindings in the sequents determine field values uniquely, so there is exactly one sequence of tokens that matches the whole pattern. That sequence of tokens is emitted by the encoding procedure. The encoding procedure also contains code that checks to ensure that the conditions associated with the disjunct are satisfied; if not all conditions are satisfied, the encoding procedure calls an error-handling procedure supplied by the application.

When conditional assembly is used or when the encoding of an instruction is underdetermined, the pattern \(p\) can contain multiple disjuncts. In these cases, the encoding procedure emits the first disjunct (sequence) whose conditions are
satisfied. It is unusual for the encoding of a single instruction to be underdetermined, but it happens on the MIPS, where the beq and bne instructions may be encoded with either 4 or 6 in the code field.

When generating encoding procedures, the toolkit checks to see that a value is specified for every bit of every emitted token. The toolkit warns of underdetermined bits, and it issues an error message for overdetermined bits. Bits are overdetermined only when a user constrains or binds overlapping fields. Range constraints on a single field are combined, and field bindings on a single field combine at encoding time to produce a single field binding, plus conditions. Finally, the toolkit uses equations to ensure that the field bindings in pattern $p$ are expressed entirely in terms of the operands $x_1, x_2, \ldots, x_n$. Any constraints found by the equation solver become conditions associated with $p$'s disjuncts; $p$ cannot be encoded unless these conditions are satisfied.

4.2 Implementing matching

It is possible to implement matching statements by checking each arm in turn to see if the input stream matches the pattern labelling that arm, stopping and executing the arm's code when the first matching pattern was encountered. This implementation is unnecessarily inefficient; it uses no information from early arms to speed up matching of later ones, and the worst-case cost of matching is proportional to the number of arms. Instead, the toolkit considers the entire sequence of arms together and generates code that identifies the matching arm. The toolkit tries to minimize the number of tests needed to identify an arm, and in the common case of linear patterns, which have no associated conditions, the matching code never tests any field more than once. The recognition takes place in two stages: in the first stage, the toolkit traverses a decision tree, testing the values of fields that appear in range constraints. In the second stage, it evaluates conditions sequentially, stopping when it reaches a pattern whose conditions are satisfied. Conditions may be part of the pattern, or they may be discovered by the equation solver.

Before working on the matching statement in toto, the toolkit converts the patterns that label the arms to a simplified normal form. In each field binding and range constraint, the toolkit tags the field with the location and width of the token containing it. The location is a bit offset from the location being matched; for example, the 32-bit displacement in an indexed-mode add instruction on the Pentium might be "the 32-bit token at an offset of 24 bits." Fields tagged with this information are called absolute fields, and the new normal form is absolute normal form, in which there are no sequences; the disjuncts contain sets of absolute field bindings and absolute range constraints.

After converting patterns to absolute normal form, the toolkit transforms the field bindings by binding each absolute field to a fresh variable. These variables are equated with the expressions from the original field bindings, and the equations are solved for the pattern's free variables. The solver returns a function for each free variable, and it may return conditions for the match. A contrived example can show how it works.
(create a single undecided node, and associate all the arms with it)
while (there is an undecided node n) do
  if (we can tell whether one of n's arms matches by checking range constraints) then
    (mark n as an internal node)
    (choose an absolute field f to test at n)
    (partition f's range according to constraints used in arms)
    (make one new undecided node, a child of n, for each element of the partition)
    (for each child c of n) do
      (associate with c the subset of n's arms satisfied by the range leading to c)
      (copy the arms associated with c and eliminate constraints of f from the copies)
  else
    (mark n as a leaf node)
Figure 10: Algorithm used to build decision tree

In the matching statement:

match pc to  
  | i32 = a + b; i32 = a - b =>
    printf("sum and difference: a = %d; b = %d\n", a, b);
endmatch
the solver would choose the variables \( t_0 \) to represent the first 32-bit token and \( t_{32} \) to represent the next 32-bit token. It would note that \( t_0 + t_{32} \) must be divisible by 2 as a condition of the match, and it would provide the following values for \( a \) and \( b \):

\[
\begin{align*}
  a &= (t_0 + t_{32}) \div 2 \\
  b &= t_0 - (t_0 + t_{32}) \div 2
\end{align*}
\]

Running the solver extracts all the information from the field bindings, which play no further role in matching. Any restrictions they may have contained are now in the form of conditions attached to patterns.

Once the solver has run, each disjunct of each pattern that label an arm is now a collection of absolute range constraints, a collection of conditions on absolute fields, and a collection of equations giving the values of free variables in terms of absolute fields. A disjunct matches if its conditions and range constraints are satisfied, a pattern matches if any of its disjuncts matches, and the toolkit must generate code to identify the first arm labelled with a pattern that matches. That identification proceeds in two stages; we use a decision tree to check range constraints and sequential evaluation to check conditions.

Figure 10 shows the algorithm used to build a decision tree. The generated matching code starts at the root of the tree and proceeds to a leaf; each node is associated with the list of arms that could match at that node. While the tree is constructed, each node is internal, a leaf, or undecided. Construction begins with a single undecided node and continues until all nodes are either internal nodes or leaves. Each internal node tests one absolute field, i.e., one field of one token. The edges leading to the node's children are labelled with sets of ranges.
Heuristic                        Score
leaffarms          1 for each child determined to be a leaf node
childarms          -1 for each arm labelling each child
nomatch            1 if each child is associated with an arm from the source
                   code; 0 if any child is associated only with the artificial
                   epsilon arm
childdisjuncts     -1 for each disjunct of each arm of each child
branchfactor       -1 for each child

Figure 11: Heuristics used to build decision trees

that partition the possible values of that field. The decision tree is compiled
into a nested case statement that tests one field after another until a leaf is
reached. At a leaf, the associated arms cannot be distinguished by range tests,
so the generated code tests the conditions of one arm after another, stopping
when it finds one whose conditions are satisfied. To ensure that every input is
matched by at least one arm of the matching statement, the toolkit adds the

| epsilon => (assertion failure)

to every matching statement.

We can judge the merit of a decision tree by counting the number of tests
required to reach a leaf. The tree with the best worst-case performance is the one
with minimum depth. Total path length might be a better measure of expected-
case performance. During construction, the fields chosen to be tested at the
internal nodes determine the shape of the tree. No polynomial-time algorithm
is known for the problem of constructing a minimum-node or minimum-path-
length decision tree, and several related problems have been shown to be NP-
complete (Comer and Sethi 1977; Hyafil and Rivest 1976). We use a set of
scoring heuristics to choose which field to test at a new internal node. Each
heuristic assigns an integer score to each field by examining the children that
would be created if that field were chosen at the current node. Not all heuristics
are used on all fields; we apply heuristic \( n \) only to the fields that achieved the
maximal score on heuristic \( n - 1 \). As soon as a single field outscores the others,
we choose that field without using any more heuristics. Figure 11 shows the
heuristics. leaffarms and childarms both promote choices that are likely to reduce
further testing later. nomatch delays tests that distinguish the special case in
which no arm matches. childdisjuncts and branchfactor favor deep, narrow trees
over shallow, wide ones.

The tree builder tests every absolute field that appears in a range constraint.
It doesn't detect that patterns like \( rs = 0 \) \( \mid rs > 0 \) always match, and therefore
for this example produces a decision tree with an unnecessary test of \( rs \).
We remove these tests by dagging, which merges isomorphic nodes and removes
internal nodes whose edges all point to the same node. Figure 12 shows the
"decision dag" that is computed for the flow-analysis matching statement in
Figure 12: Decision logic used in SPARC flow analysis

Figure 5. In this particular case, the leaheuristic by itself is enough to find a decision algorithm that reflects the hierarchical organization of the SPARC instruction set.

5 Context, Extensions, and Conclusions

We hope the toolkit can be applied to a variety of problems beyond simple encoders and decoders, both in the machine-code domain and in other problem domains. We finish by discussing some such problems, by showing how the toolkit relates to other work, and by judging the value and contributions of the toolkit.

5.1 Specifying other properties of instructions

Our machine-description language is limited to "syntactic" properties, e.g., the correspondence between assembly code and machine code. Most machine-description languages describe semantic properties. Semantic information is often complex and hard to reuse, so it is common to invent a new description language for each application. Proebsting and Fraser (1994), Davidson and Fraser (1980), and Bradlee, Henry, and Eggers (1991) present specialized machine-description languages of varying complexity.

Some problems can be solved by "syntactic" information alone, and other problems are easily solved by hand given the ability to manipulate instructions symbolically. Many important problems, however, are best solved by application generators that use semantic properties. Such properties include register-transfer semantics, which can be used to generate an instruction selector or
a program slicer, pipeline-hazard information, which can be used to generate part of an instruction scheduler. There are other properties, which are not normally considered "semantics," but which could play a similar role. Such properties include assembly-language syntax, which could be used to generate an assembler and disassembler, and models of dependencies and probabilities, which could be used to generate customized compression and decompression tools (Danskin 1994).

We have identified two strategies used in application generators that manipulate machine instructions. The first strategy is to use a machine description that contains only semantic information, to work with symbolic representations of instructions, and to use assembly language. This strategy is used by BURG-based code generators, for example (Fraser, Henry, and Proebsting 1992; Fraser and Hanson 1995). The second strategy is to use a machine description that combines syntactic and semantic information in a single description. This strategy is used by EEL, for example (Larus and Schnarr 1995). We use a similar strategy to generate the assembly-emitting procedures used in our specification checker; the Pentium description includes a description of assembly-language syntax as well as of instruction encoding.

Our descriptions of assembly-language syntax are divided into three parts: the names of the constructors, the names of the operands, and the syntactic sugar associated with the constructors. We specify the assembly-language names of constructors by giving rules for rewriting the full names to assembly-language names. Our Pentium specification follows the architecture manual in using suffixes like .Eb.Gb to distinguish opcodes that implement the same operation on values of different types. These suffixes are not used in assembly language, and our rewrite rules eliminate them. Similarly, long suffixes like .I32 and .I16 are written into 1 and s, which are the suffixes used in the assembly language. Here are a few of the rewrite rules used to make our Pentium syntax acceptable to the GNU assembler:

```
assembly opcode
{.I32,.R64,.lsI32,.lsR64} is 1
{.I16,.R32,.lsR32} is s
B.{Eb.1,Eb.CL,Eb.1b} is B
P.ST1.ST is P
{CMPXCHG,ADD,XCHG}.Eb.Gb is $1b
{IN,INT,J}.* is $1
{TEST*}. is $1
{Jv,lb} is J
{INC}.Eb is INCb
{MOVib,MOV.Eb.1b} is MOVb
TEST.Ed.Id is TEST1
```

The left-hand-side syntax is chosen to resemble csh "globbing" syntax.

We specify syntaxes for integer operands by giving printf-like format strings used to print them. The %s format specifies the use of field names for values; many architectures refer to registers by name, not by number.
The following rules suffice to express operands on the Pentium:

```
assembly operand
[i8 i16 i32] is "$/.d"
[x32 x16 x8 base index] is "$%v"
[reg reg8 sreg cr dr] is "$%v" using field base
```

Since reg, reg8, etc. are not fields, we specify which field should be used to derive names for them. The names of fields are given by a `fieldinfo` specification like that shown on page 11:

```
fieldinfo [ base index ] is
[ names [ eAX eCX eDX eBX eBP eSI eDI ] ]
```

Finally, we specify syntaxes for constructors using the form of specification that appears on the left-hand side of the full constructor definitions. The operand names must match the names used in the original definition. On most machines, it would be unnecessary, since the original constructor definitions provide sufficient information about assembly-language syntax, but there are several incompatible assemblers used on the Pentium. Our original description uses the syntax suggested in the architecture manual, and we needed a different syntax to build a checker using the GNU assembler. For example, as shown in Figure 1, the Intel manual suggests a syntax of `d(base,index.ss)` for the Index32 addressing mode, but to satisfy the GNU assembler, we must specify the syntax as follows:

```
assembly syntax
Index32 d(base,index.ss)
```

This specification technique makes it possible to change not just the syntactic sugar surrounding the operands, but also their order.

Our specifications of assembly-language syntax use only the names of the constructors and their operands. Other kinds of semantics, like those used for instruction selection or scheduling, might also be specified using these names. We would like to see this observation exploited to create a third strategy for building application generators. This strategy would use multiple machine descriptions, one for each aspect of the problem that needed to be solved. We hypothesize that such descriptions would use constructors and operands in common. We don’t know whether there are other specifications elements that should also be used in common. Given a collection of related descriptions, and given a collection of components each of which “understands” one kind of description, we would like to build application generators by composing these components. To facilitate investigation of these questions, we are rewriting the toolkit in Standard ML (Milner, Tofte, and Harper 1990).

5.2 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 es-
sentially a macro processor with bit-manipulation operators and special support for different integer representations.

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. We prefer to build applications by using several simple specifications, each describing different properties of the same machine, to build different parts.

The GNU assembler provides assembly and disassembly for many targets, but different techniques have been applied ad hoc to support different architectures (Elsner, Fenlason, 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. Their machine descriptions has a syntactic part, which resembles and is derived from the 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 determines classifications, such as 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 pre-encoded "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 (ISO 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.

5.3 Future work

It would be instructive to experiment with multi-pass strategies for conditional assembly, to make conditional assembly an efficient, machine-independent way of specifying how to assemble span-dependent branches. Such strategies can change the size of previously emitted instructions. Size changes could be accommodated by putting each varying instruction in its own relocatable block, but it would be awkward to expose these extra relocatable blocks to an application.

One could store relocatable blocks and relocation closures in a file and use the collection as a machine-independent representation of object code. Such an object-code format could make it easier to write testing tools like Purify (Hastings and Joyce 1992), profilers and tracers like qpt (Ball and Larus 1992), and optimizing linkers like OM (Srivastava and Wall 1993), all of which manipulate object code. One could add a machine-independent linker to the toolkit’s library and extend the generator to generate assemblers from specifications, making it possible to take assembly code generated by existing compilers and assemble it into this new format. The only nontrivial part of this approach is externalizing the function pointers contained in the relocation closures. Ramsey (1995a) describes two possible solutions to this problem and a prototype implementation.

We believe the ideas in the toolkit can be exploited to work with other forms of binary-encoded data, not just machine instructions. We are designing an extension to the toolkit that enables it to describe arbitrary sequences. Such sequences are common components of network messages, and this extension would make it possible to use the toolkit to generate encoding and decoding code for such messages. If the toolkit’s specification language were integrated with a compiler, one could tell the compiler to provide a “view” of arbitrary binary data as an ordinary structure or union in a high-level language. Such a facility would be useful in projects like the Fox networking project (Biagioni et al. 1994) and the SPIN operating system (Hsieh et al. 1995).

We are also investigating the use of toolkit specifications to help build compression models for arithmetic coding (Bell, Cleary, and Witten 1990). It may be increasingly useful to compress machine code as the gap between processor speeds and secondary storage widens. Better compression would help reduce the storage requirements for large network traces, which can consume gigabytes (Duffy et al. 1994).

5.4 Evaluation

The toolkit is intended less to support traditional compilation than to support nontraditional operations like rewriting executable files or run-time code gener-
In fact, there are two ways in which the toolkit is currently less suitable for traditional compilation than simply using the vendor's assembler: it can't currently represent relocatable object code on disk, and it can't make use of optimizations that vendors may build into assemblers, like MIPS instruction scheduling.

Our specification language 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 into 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 language sufficed to describe all of the MIPS and most of the SPARC, and we used it to generate encoding procedures for mid. 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 the language described in this paper.

Patterns are a simple yet powerful way to describe the binary formats of instructions. Field constraints, conjunction, and concatenation are all found in architecture manuals, and together they can describe any instruction on any of the three machines we have studied. They're not limited to traditional instruction sets in which opcode and operand are clearly separated; all three machines 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.

If patterns provide a good high-level description of binary encodings, constructor specifications raise the level of abstraction to that of 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 constraints, which are part of the definitions of some architectures, like the Intel Pentium.

We maximize the power of the toolkit's specification language 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 parts of the language 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.

Acknowledgements

This work has been funded by a Fannie and John Hertz Fellowship, an AT&T PhD Fellowship, an IBM Graduate Research Fellowship, and by Bellcore.

Availability

The current version of the toolkit is version 0.3. It is available by anonymous ftp from ftp.cs.princeton.edu in directory pub/toolkit. The toolkit also has a home page on the World-Wide Web; the URL is http://www.cs.princeton.edu/software/toolkit.

Production notes

We prepared this paper using the noweb tools for literate programming (Ramsey 1994b). The SPARC examples have been checked for consistency with Appendix A. All of the examples have been extracted from the paper and run through the toolkit, and they all work with version 0.3. Figure 12 was generated automatically from the code in Figure 5.

References


Fraser, Christopher W., David R. Hanson, and Todd A. Proebsting. 1992 (September). Engineering a simple, efficient code-generator generator. ACM Letters on Programming Languages and Systems, 1(3):213–226.


A Partial SPARC spec

This partial specification of the SPARC includes all the examples in the paper. It omits many floating-point instructions, several flavors of load, store, read and write instructions, and many synthetic instructions. The reader is encouraged to compare the specification to the equivalent information provided in the SPARC architecture manual (SPARC 1992). Page references are provided for cross reference.

This fields declaration defines the fields used in all SPARC instructions:

fields of itoken (32)
op 30:31 rd 25:29 op3 19:24 rs1 14:18
i 13:13 imm13 0:12 opf 5:13 rs2 0:4
op2 22:24 imm22 0:21 a 29:29 cond 25:28
disp22 0:21 asi 5:12 disp30 0:29

The following patterns represent Tables F-1 and F-2 on p 227.

patterns
TABLE_F2 & op2 = {0 to 7}
TABLE_F4

The following patterns represent Table F-3 on p 228.

patterns
TABLE_F3 & op3 = {0 to 63 columns 4}
The following patterns represent Table F-4 on p 229.

```plaintext
( ld lds ldd ldsb ldsh ldub lduh ldcld df stdcld
" stfar stdfq stdcq
stb stda stdf stdc

ldab ldsba ldsb ldsba
ldsh ldsha lduha ldsh
ldstub ldstuba

swap swapa

TABLE_F4 / op3 

The unimp pattern is used as a place holder in the instruction stream for instructions that refer to unknown relocatable addresses.

The u.oimp pattern is used as a place holder in the instruction stream for instructions that refer to unknown relocatable addresses.

Address operands, defined on 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

Register or immediate operands, defined on p 84, have two possible formats.

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

The following example specifies the assembly syntax and binary encoding for all load-integer instructions defined on p 90. It shows that Address operands are delimited by brackets in the assembly language.

 Patterns loadg is ldab | ldsh | ldub
  | ldsb | ld | ldd

 Constructors
  loadg [Address], rd
```

46
The assembly syntax and binary encoding for all alu instructions is the same.

patterns
logical is and | andcc | andx | andncc
| or | orcc | ora | orncc
| xor | xorc | xor | xorcc
shift is sll | srl | sra
arith
is add | addcc | addx | addrc | taddcc
| sub | subcc | subx | subcc | tsubcc
| umul | umulcc | umulcc | umulcc | umulcc
| udiv | udivcc | udivcc | udivcc | udivcc
| save | restore | taddctv | tsubctv
alu is arith | logical | shift

The following patterns group the logical, shift, and arithmetic opcodes.

patterns
logical is and | andcc | andx | andncc
| or | orcc | ora | orncc
| xor | xorc | xor | xorcc
shift is sll | srl | sra
arith
is add | addcc | addx | address | taddcc
| sub | subcc | subx | subcc | tsubcc
| umul | umulcc | umulcc | umulcc | umulcc
| udiv | udivcc | udivcc | udivcc | udivcc
| save | restore | taddctv | tsubctv
alu is arith | logical | shift

The assembly syntax and binary encoding for all alu instructions is the same.

constructors
alu rs1, reg_or_imm, rd

The following pattern represents the first column of Table F-7 on p 231.

patterns
branch is any of
[ bn bne ble bl bleu bcs bneg bvs
ba bne bg bge bgu bgeu bpos bvc
which is Bicc & cond

where
p is any of [a b ... z],
which is generating expression
is syntactic sugar for
[a b ... z] is generating expression
p is a | b | ... | z

The synthetic instructions bset and dec are defined on p 86. They are assembled
using the machine instructions or and sub.

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

The assembly syntax for branch instructions is defined on pp. 119-120. The
fieldinfo declaration makes the compound names come out right.

fieldinfo a is [names ["", ",a"]]
relocatable addr
constructors
branch\*a addr { addr = L + 4 • disp22! } is L: branch & disp22 & a

The conditionally assembled instruction set is defined on p 84. This definition
tries to assemble set into a single instruction when possible.

constructors
sethi val, rd is sethi & rd & imm22 = val[10:31]
set val, rd
when { val[0:9] = 0 } is sethi(val, rd)
when { val = val[0:12]! } is or(0, imode(val), rd)
otherwise is sethi(val, rd); or(rd, imode(val[0:9]), rd)
# Contents

1 Introduction ............................................. 2

2 Elements of Specifications and Applications .............. 4
   2.1 Tokens and fields .................................. 5
   2.2 Patterns ........................................... 6
   2.3 Constructors ....................................... 7
   2.4 Relocation ......................................... 8
   2.5 Synthetic instructions and conditional assembly ...... 9
   2.6 Matching statements .................................. 10

3 Using Specifications and the Toolkit ...................... 12
   3.1 Using the toolkit in applications .................... 13
   3.2 Naming issues ....................................... 14
   3.3 Specification checkers ............................... 15
   3.4 Implementation ..................................... 16

4 Creating Encoders and Decoders ......................... 18
   4.1 Representing patterns and constructors ............... 19
   4.2 Representing and encoding constructors .............. 20

5 Context, Extensions, and Conclusions .................. 22
   5.1 Specifying other properties of instructions .......... 23
   5.2 Related work ....................................... 24
   5.3 Future work ........................................ 25
   5.4 Evaluation .......................................... 26

References .................................................. 27

A Partial SPARC spec ...................................... 28