New Jersey Machine-Code Toolkit Reference Manual Version 0.5a

Norman Ramsey   Mary Fernandez

February 20, 2001

Contents

Chapter 1

Introduction

This is the reference manual for version 0.5a of the New Jersey Machine-Code toolkit. New users should read our overview (Ramsey and Fernández 1995) and study our annotated descriptions of real machines (Ramsey and Fernández 1994) before tackling this manual. Our long paper on the specification language (Ramsey and Fernández 1997) is also worth reading. Here you will find:

Chap. 1	Promises, cautions, and warnings.
Chap. 2	A grammar for the specification language, with a detailed description of each construct.
Chap. 3	How to write decoding applications. Describes matching statements.
Chap. 4	How to write encoding applications. Describes the toolkit library, which provides support for emitting data into streams and relocating instructions.
Chap. 5	How to run the generator and translator. Describes all their command-line options.
Chap. 6	How to check the consistency of a specification using a vendor's assembler and disassembler.

1.1 Caveat user

Version 0.5a has been exercised more heavily than other versions, but we believe that errors linger in both the manual and the code. If you have a problem with the toolkit, please tell us. That's the only way we can make it right. Send email to `toolkit@cs.princeton.edu`.

1.2 Discussing the toolkit

Starting with the release of version 0.5a, the mailing lists related to the toolkit have moved (with Ramsey) to Harvard University. `toolkit-interest@lists.eecs.harvard.edu` goes to people who have expressed some interest in the toolkit, and it carries most general announcemnets about the toolkit. `toolkit-users@lists.eecs.harvard.edu` goes to people who are actually using the toolkit, and it carries discussion among users. It's still a good idea to report problems to `toolkit@cs.princeton.edu`.

1.3 Multiple-language support

The toolkit currently has rudimentary support for multiple languages. For decoding applications, this support actually works; we use it both for C and for Modula-3. The support for encoding applications is only partly in place; there are lots of little C-isms lurking around the declarations of encoding procedures and the results of typed constructors, and there is only one version of the toolkit's library, which is written in C. We would be pleased to try to support users interested in other languages. We would probably ask such users to develop the library support, since that is a fairly straightforward and separable task. We would tackle the details of making the generator truly support multiple languages.

1.4 Known bugs

Almost all of the open bugs that actually bit people have been fixed in version 0.5a. We still have plenty of open bugs that nobody has complained about yet. Pester us if you want them fixed.

- Field-width conditions don't show up in closures.
- Generated decoders assume that input streams go on forever—they have no way of coping with (or even indicating!) end of stream.
- We don't implement the obvious optimization that avoids creating closures for branches into the current relocatable block.[1]
- The equation solver is shaky, particularly around bit slices. You should be OK if you make sure your slices partition the bit space—don't allow overlaps or gaps.
- We don't solve the MIPS `li` constructor on decoding because the solver isn't quite bright enough to realize that if it knows `n@[0:15]`, it also knows `n@[15:15]`. We'll work on it. Meanwhile, there's a workaround: insert the equation `dummy = imm@[0:14]` in all three branches of the constructor.

[1] OK, a missing feature is not a bug. But it should be there.

- The toolkit dies on the first error, and our error messages are terrible. In particular, all messages are treated using the syntax-error mechanism, which means that semantic errors point to the end of the specification file in the most misleading way. Using the `-impossible` option sometimes gives enough information to help track down the real source of an error.

  Real improvement in error handling will probably await re-implementation in a language better suited to the problem (like one that has exceptions!).

1.5 Change history

1.5.1 Changes in version 0.5a

- Better generation of disassemblers.
- Toolkit library no longer depends on [[stdout]] being a static value.
- Other bug fixes lost in the mists of time.

1.5.2 Changes in version 0.5

- The distribution includes a specification of the Alpha, which we have tested against Digital's assembler.
- The toolkit no longer generates dead code in decoders. On the other hand, it also generates more common subexpressions.
- Field conditions are now used to distinguish branches. In a multi-branch constructor, if a value won't fit in the field to which it is bound, the toolkit goes on to the next branch. One consequence is that warnings of "redundant disjuncts" should now be taken much more seriously.
- We now permit one-integer specifications of one-bit fields, by analogy with one-integer specifications of one-bit slices.
- The toolkit issues a warning if it doesn't see the [[expects in fetch templates.
- Fixed a bug in which generated decoders sometimes referred to variables before they were declared.
- Fixed a bug that prevented decoders from working when constructor types were nested, as on the Pentium.
- Fixed a flagrant bug in the big-endian bit numbering.
- Removed some redundant checks from generated encoding procedures.
- Closures are fewer and more efficient.

1.5.3 Changes in version 0.4

The most notable change in version 0.4 is support for checking the consistency of a specification against a vendor's assembler and disassembler. See Chapter 6 for a description of the checker.

- The toolkit can now validate a specification against an existing assembler (Chapter 6).
- We have validated our MIPS, SPARC, and Pentium specifications. We found and corrected a handful of errors.
- The manual now describes assembly syntax specifications (Section 2.10). (Apparently it doesn't yet say how to use them?!)
- The examples directory now includes a rudimentary disassembler for the SPARC. Thanks to Cristina Cifuentes for providing the impetus.
- It is now possible to use `_` in expressions as a true "wildcard" value. Unlike in previous releases, different instances of `_` are really different, even if they appear in the same arm of a matching statement. It works by representing a different, otherwise unnameable variable at each instance. Suggested uses include:
  - Use it in matching statements for uninteresting arguments to constructors.
  - To constrain `n` to be a multiple of 4, write the equation `n& = 4 * _`.
- We have better diagnostics to help you figure out what is wrong when you write a matching statement and some arms never match.
- We have added a few procedures to the toolkit's library.
- We have fixed bugs:
  - We now believe that constructor names in matching statements (using the `[name]` binding statement) are bound correctly.
  - If the placeholder `_` is used multiple times on the left of a matching statement, each instance is now distinct.
  - Nested applications of constructors should work correctly in matching statements.
  - The toolkit used to generate bad code when there were conditions on decoding—some variables were used before they were declared or initialized. We believe we fixed this bug on December 14, 1995.
  - And many more...

1.5.4 Changes in version 0.3

- Field bindings can now be full expressions. As a result, the syntax for bit slicing has changed.
- Patterns can now include labels. As a result, the "`$pc` hack" is no more.
- Dramatic improvements in our handling of closures; see (Ramsey 1996).
- Eliminated blanket warning about field-name literals in favor of warnings about particular literals that could cause trouble.
- In decoders, generate if statements instead of switch statements where reasonable, and prefer equality tests to range tests.
- Support for assembly-language syntax, encoders that emit assembly, and preliminary support for generating assembly-language grammars.
- Other changes and improvements too numerous to mention.

1.5.5 Changes in version 0.2

- The toolkit can now generate encoding procedures that emit assembly language. New assembly-syntax specs can change form of this assembly language.
- It is now legal to use expressions, not just identifiers, in field bindings and constructor applications.
- To support wider use of expressions, we were forced to change the syntax of the bit-slicing operation from, e.g., `n[0:4]` to `n@[0:4]`.
- We now support unary minus in expressions.
- `$pc` is gone, having been replaced with a more general mechanism: pattern labels.
- The toolkit now supports both little-endian and big-ending bit numbering (but big-ending hasn't been tested).
- Integer literals can be written in octal or binary formats as well as decimal and hexadecimal.
- Many bugs have been fixed. Many more remain.
- We changed the precedence of the pattern operators so it more accurately reflects the normal form.

Chapter 2

The specification language

This chapter describes the language that the toolkit uses to describe the encodings of machine instructions. We write a grammar in which *nonterminals* appear in italics, *terminals* appear slanted, and `keywords` and literal text appear in typewriter font. Terminals ending in *-name* are identifiers. Terminals with no special names are integer literals unless otherwise stated.

To describe the specification language, we use an EBNF grammar. We use the standard EBNF metasymbols for

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

We use large metasymbols to make them easy to distinguish from literals.

The start symbol for the grammar describing a specification is *Specification*. Definitions must appear before uses, but otherwise the parts of a specification may appear in any order. The grammar for a complete specification is therefore

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

2.1 Tokens and fields

The specification

$$\textit{spec} \Rightarrow \texttt{fields of}\ \textit{token-name}\ \texttt{(}\ \textit{width}\ \texttt{)}\ \textit{field-specs}$$

$$\textit{field-specs} \Rightarrow \big\{\textit{field-name low-bit}\texttt{:}\textit{high-bit}\big\}$$

defines a token named *token-name* that is *width* bits wide, as well as all the fields that may be used to refer to parts of that token. Bits are numbered from 0. By default, bit 0 is the least significant bit, but the significance of bit 0 can be changed with the `bit 0` directive (Section 2.9.2).

A `fieldinfo` specification gives auxiliary information about a field or list of fields. (It is useful to describe several fields simultaneously when, for example, they all refer to the same set of registers.) Such auxiliary information might include special names used to refer to values of the fields (as when they describe data formats or registers) or information about what kind of checking should be done when an application supplies a putative value for a field.

The syntax of `fieldinfo` specifications is:

spec ⇒ `fieldinfo` {*fieldinfo-spec*}

fieldinfo-spec ⇒ *field-specifier* `is` `[` {*field-item*} `]`

field-specifier ⇒ field-name | `[` {field-name} `]`

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

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

The terms `checked`, `unchecked`, and `guaranteed` denote three possible levels of checking in generated encoding procedures:

- If f is a checked field, the encoding procedure checks to make sure its value falls within range. If not, it calls an error handler.
- If f is an unchecked field, the encoding procedure does not check its value, but it does mask out high bits, ensuring that other parts of the token can't be corrupted by a bad value of f.
- If f is a guaranteed field, the encoding procedure simply uses its value without any checking or masking at all.[1] This option is useful mainly for situations in which f denotes a register, and the register allocator promises never to use a bogus register.

`sparse` and `names` are both ways of specifying names of fields. `names` is used when all the names are known; `sparse` is used otherwise. Identifiers can be used to specify names on most targets, but names containing special characters can only be specified using string literals. Note that the toolkit accepts the C literal syntax for characters in place of integer literals; this property may be useful when writing `sparse` bindings. The use of commas to separate lists of bindings is not consistent with our other representations of lists, but they look funny without the commas.

2.2 Patterns

Patterns can be defined both in terms of themselves or in terms of simpler elements. We discuss the base cases of the recursion first.

[1] Signed field parameters must always be masked, but most field parameters are unsigned.

2.3 Simple patterns

Simple patterns can be described by any of the following syntaxes:

pattern ⇒	*name*	*name* of pattern, field, or constructor type
|	*opcode* `(` *arguments* `)`	Constructor application
|	*field-binding*	Binds field to variable
|	*constraint*	Constrains field
|	`epsilon`	The empty sequence.
|	`some` *token-name*	Matches a single token of the class named.
|	`[` {name} `]`	List of patterns

field-binding ⇒ *field-name* `=` *expr*

constraint ⇒ *field-name relational-operator* (*integer* | *generating-expression*)

relational-operator ⇒ `<` | `<=` | `=` | `!=` | `>` | `>=`

Unless a list of pattern names or a generating expression is used, the pattern denotes a single pattern; otherwise it denotes a list. Lists are ordinarily used to describe opcode tables. The *arguments* in a constructor application may include expressions, literal strings (which must be field names), or other constructor applications. Section 2.8 describes the syntax of expressions (*expr*). Expressions in a *field-binding* may contain free variables. Such variables may be operands, or they may be computed by the solver.

When an identifier is used as a pattern, its meaning depends on context. Pattern names defined with the `patterns` pattern-binding statement can always be used in later patterns. On the right-hand side of a constructor definition, the names of operands that represent fields or constructor types may also be used as patterns. The name of a field operand `f` is short for

`f` `=` *the value of the operand* `f`.

Similarly, the name of a constructor-type operand `T` is short for its value, which must be a pattern because the result of applying a constructor is always a pattern.

The *opcode* used in a constructor application may denote a single constructor or a group of constructors with identical operands. It works the same way in a constructor specification, as described in Section 2.7. Strictly speaking, applying a constructor to create a pattern is not "simple," because patterns themselves are used to define constructors. It is actually part of a mutual recursion between the definitions of patterns and constructors.

A *generating expression* can be used instead of an integer in field constraints. The resulting constraint denotes a list of patterns, one for each value of the generating expression. There are three forms of generating expression:

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

The first form generates the integers from lo to hi inclusive, in order. The second form generates the same integers, but in an order suited to an n-column layout, e.g. $lo, lo+n, lo+2n, \ldots$, wrapping around as needed. The third form simply lists the integers generated; it is suitable for sparse tables.

2.4 Combining patterns

The recursive part of the definition of patterns makes it possible to combine patterns using the following operators:

pattern $\Rightarrow$	*pattern* `...`	Loosens rules restricting conjunction.
$\mid$	`...` *pattern*	Loosens rules restricting conjunction.
$\mid$	*pattern* `&` *pattern*	Conjunction.
$\mid$	*pattern* `;` *pattern*	Sequence.
$\mid$	*pattern* `|` *pattern*	Disjunction.

Operators listed first have higher precedence.

2.5 Pattern labels

One can refer to a location within a pattern by using a *pattern label*. The syntax is

pattern $\Rightarrow$ *label-name pattern*

The appearance of *label-name* is a binding instance; its scope is the `patterns` declaration or the constructor branch in which it appears. Within that scope, it refers to the location in the instruction stream immediately preceding the pattern it labels. That location can be converted to an integer in the same way as a relocatable address. Labelling has the same precedence as the sequencing operator, so, for example, to refer to the location at the end of an instruction, one can use the idiom

pattern`; L: epsilon`

2.5.1 Internal form of patterns

Patterns constrain both the way in which streams are divided into tokens and the values of the fields in those tokens. Internally, patterns are composed from *range constraints* and *field bindings*. A range constraint limits the acceptable values of a field to a range known at compile time. Often this range contains only a single element, e.g., as fixed by the constraint `op = 1`. A field binding forces a field to have a particular value, but that value may not be known at compile time—in general, it depends on the values of operands passed to a constructor. The binding `offset = (target - L)[2:17]` is an example.

The toolkit represents range constraints in the form $lo \le f < hi$. The toolkit uses its knowledge of field widths to map relational expressions into range constraints. For example, `f = 9` maps into $9 \leq f < 10$, and `f > 3` maps into $4 \leq f < 2^{width(f)}$.

The toolkit represents field bindings in the form $f = expression$. Expressions may be written explicitly, or they may be identifier's whose values are determined by the toolkit's equation solver. Depending on whether an application does encoding or decoding, the solver may compute field values as a function of constructor operands or vice versa.

Range constraints and field bindings form the basis for patterns, which are represented internally in "disjunctive" normal form. Patterns are disjunctions. Each disjunct is a sequence. Each sequent applies to tokens of a particular class, and it is a (possibly empty) collection of constraints of fields in that class.[2] Many interesting patterns are single disjuncts; for example, `epsilon` is the empty sequence, and `some` *class-name* is a sequence with one element (sequent), that element containing no constraints. The pattern `f = 0` is a sequence with one element, that element containing the single range constraint $0 \leq f < 1$. The inequality operator is an exception to this happy rule. In general, an inequality can't be represented as a single range constraint, so `f != n` is syntactic sugar for a disjunction:

$$0 \leq \mathtt{f} < \mathtt{n} \mid \mathtt{n} + 1 \leq \mathtt{f} < 2^{width(\mathtt{f})},$$

Since a disjunction almost never makes sense when encoding, using inequality constraints is asking for trouble. We get away with it in a few cases where `n` is zero, in which case one of the disjuncts is vacuous and is discarded.

2.5.2 Pattern matching and the meanings of the operators

The basic patterns and pattern operators can be described in terms of their effect on the normal form, but it is easier to consider how they affect matching. The constraint $lo \leq \mathtt{f} < hi$ matches an input token if the `f` field of that token falls in the range defined by lo and hi. `f` is declared as a field of a particular class of tokens, which determines the size of the token matched. The wildcard 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. A concatenation "`p; q`" matches if `p` matches an initial sequence of input tokens and `q` matches the following tokens. A disjunction "`p | q`" matches if either `p` or `q` matches.

2.5.3 Pattern shapes and the ellipsis

Concatenation and conjunction distribute over disjunction, so it is usually convenient to talk about operations on disjuncts (sequences). Each disjunct (sequence) has a unique *shape*, which is obtained by taking the class of each element of the sequence. So, for example, on the Intel 486, a 32-bit displacement addressing mode defined by the pattern

[2] By "constraint" we mean either a range constraint or a field binding.

```
mod = 2 & r_m != 4; i32 = d
```
[3]

has the shape `modr_m; I32`.

The toolkit uses shapes to restrict conjunctions; two patterns may be conjoined if and only if they have the same shape. For example, on the 486, the pattern `mod = 0 & r_m = 5` is permitted, but the pattern `mod = 0 & index = 2` is not. The latter conjunction attempts to combine patterns of shape `modr_m` and `sib`.

The ellipsis operator provides a way to loosen this restriction. "`p ...`" creates a pattern that is equivalent to `p`, except it is OK to write `p ... & q` whenever `p`'s shape is a prefix of `q`'s shape. Examples can be found in the description of the Intel 486 by Ramsey and Fernández (1994). The ellipsis may also be used as a prefix operator on patterns, in which case `... p & q` is permissible whenever `p`'s shape is a suffix of `q`'s shape. We haven't had occasion to use the prefix ellipsis in machine descriptions, because most hardware decodes complex instructions from left to right.

Be careful to surround the ellipsis operator with whitespace. The dot is just like a letter for purposes of lexical analysis, so

```
p...
```

is a four-character identifier name, not the pattern `p` followed by the ellipsis.

2.6 Pattern bindings

Patterns can be named using pattern-binding declarations:

spec ⇒ `patterns` {*pattern-binding*}

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

The first form names a single pattern.

The second, most common form binds a list of names on the left to a list of patterns on the right. The number of names must match the number of patterns; the special name "`_`" can be used to match uninteresting patterns on the right. This form requires the use of generating expressions (or an explicit list of patterns) to produce a list of patterns. It is typically used to create opcode tables. In this situation, the name "`_`" simply indicates an unused opcode.

The third form extends the second to handle a common idiom. The sequence of pattern names and the pattern on the far right act just as they do in the second form, but in addition, the single name on the left is bound to the disjunction of all the patterns on the right that are bound to names other than "`_`". Examples can be found in the SPARC and 486 specifications given by Ramsey and Fernández (1994).

[3] The `!=` relational operator is a cheat. Don't use it.

The name to which a pattern is bound remains associated with that pattern, and that name may be used in the formation of opcode names. When a disjunction is bound, the individual disjuncts retain their names, if any. This behavior can be exploited when writing constructor specifications, in which disjuncts are enumerated when patterns appear in opcodes.

2.7 Constructors

The constructor specifications are the most complicated in the toolkit; they can use compound opcodes, equations, assembly-language syntax, and conditional assembly. Constructor specifications are the only ones in which newlines are significant. We make newlines significant so that we can use them in the very common case in which the output pattern of the constructor is implicit.

The syntax of constructor specifications is

$$\textit{spec} \Rightarrow \texttt{constructors}\ \{\textit{constructor}\}$$

$$\textit{constructor} \Rightarrow \textit{opcode}\ \{\textit{operand}\}\ [\ :\ \textit{type-name}\ ]\ \star\ [\textit{branches}]$$

Because both type and branches are optional, we use the newline to separate the last operand of the current constructor from the opcode of the following constructor. The newline is significant only in the position marked $\star$, and only the first newline is significant. It is OK to include the newline even when explicit branches are present.

Constructors specified by the toolkit are *typed* or *untyped*, depending on whether a *type-name* appears. All constructors are applied to operands to produce patterns, but the encoding procedures that the toolkit generates for a constructor have different effects depending on typing. Untyped constructors emit tokens into the current instruction stream, but typed constructors produce opaque values encapsulating patterns. The values are opaque in the sense that their only use is to be passed as operands to other constructors.[4] A new constructor type is defined simply by using a new *type-name* in a specification. All constructors of a type must be defined before the type is used, for example, as an operand to another constructor. This restriction prevents circular definitions.[5]

The opcode and operand notations used in constructor specifications are designed to enable concise specifications that resemble assembly language. In fact, the toolkit uses these notations to infer an assembly-language syntax for each instruction, and the toolkit can generate encoding procedures which emit

[4] We recognize that this restriction is a defect in the toolkit, and that application writers might like to match on or manipulate the opaque values produced by typed constructors. We welcome suggestions about how such a facility should be designed, implemented, or made available to applications.

[5] It also prevents recursive constructor types. We avoid recursive constructors for two reasons. For efficiency, the toolkit's generator "inlines" constructor operands by enumerating all possible patterns, a technique that works only when the enumeration is finite. When decoding, the toolkit can use a simple recognition algorithm instead of having to create a parser, which might be ambiguous.

that assembly language. As noted above, the constructor specification begins with opcode and operands. The list of operands is terminated by the first newline, or by any of the tokens that introduce later constructs.[6]

Opcodes are formed from three kinds of units: literal strings, pattern names, and field names. Literal strings simply contribute to the opcode name, but patterns and fields introduce an implicit iteration: patterns iterate over disjuncts, and fields iterate over named values. Each iteration results in the specification of a separate constructor. For a pattern, the name of the disjunct is substituted for the pattern name in the opcode, and the disjunct is used to stand for the pattern on the right-hand side. At the moment, that value isn't used properly in equations. This is an outstanding bug; the value should be equivalent to an integer literal in the equations. For a field, the name of the field's value[7] is substituted for the the field name on the opcode, and the pattern binding the field to that value is used to stand for the field on the right-hand side. Any pattern so introduced on the right-hand side is conjoined into the *implicit output pattern*, which is the one used in the constructor specification when branches are omitted. For examples, see Ramsey and Fernández (1995). The basic idea is for any pattern or field on the left to be re-used on the right — typically it will be conjoined in, in which case it is used implicitly.

The elements of an opcode are joined with a hat (`^`). The syntax for an opcode is

$$\textit{opcode} \Rightarrow \textit{opname}\ \{\hat{}\ \textit{opname}\}$$

$$\textit{opname} \Rightarrow \textit{string}\ \big|\ \textit{pattern-name}\ \big|\ \textit{field-name}$$

Most opcodes use just a single name. It's permissible to use an unbound identifier as an *opname*, which has the same effect as a literal string.[8]

Our operand syntax is unusual in that we permit arbitrary "noise words" to enable the specification writer to mimic assembly-language syntax. We call such noise words *literals*. We use literals to help create encoding procedures that emit assembly language. For binary encoding and decoding, literals are ignored completely. We permit any integer or quoted string, plus a variety of special characters that are common in assembly languages but don't conflict with our usage in specifications:

literal ⇒ *string* | *integer*
| any of these characters: `<>=[]()+-/&@#%;*$,|`

[6]Computers are better than I at computing follow sets, but certainly the colon that introduces a constructor type, the brace that introduces equations, and the keyword `is` all terminate lists of operands.

[7]As given by `names` or `sparse` in a `fieldinfo` declaration.

[8]This is hokey language design, but it makes the specifications look nice. It's unlikely to lead to errors, because a misspelling, which would result in an unbound identifier, would also result in that identifier's being unbound on the right-hand side, which would be flagged as an error. (We don't expect users to write specifications in which pattern or field names are used on the left without also being used on the right, and we issue a warning message if they do.)

The real operands of a constructor are all given by name. Constructor operands may have several different types, but we don't want to clutter up our notation with type information, so we use special names shamefully:

operand ⇒	*field-name* [`!`]	Field, possibly signed
|	*constructor-type-name*	Result of applying typed constructor
|	*relocatable-name*	Relocatable address
|	*unbound-name*	Integer operand
|	*literal*	Ignored for encoding and decoding

Constructor inputs may have underscores and digits appended; they can be used to distinguish multiple inputs of the same type. Field and constructor inputs are available to be used as output patterns; fields stand for the pattern binding that field to the value of the operands, and constructor inputs stand for themselves—the result of applying a constructor is always a pattern. Relocatable addresses and integers can be used in equations, in field constraints, or both. A *relocatable-name* is an identifier declared with the `relocatable` directive described in Section 2.9.5.

The toolkit always treats field values as unsigned. When a field is actually signed, specification writers must use an exclamation point as an explicit sign-extension operator. Indulging in a minor abuse of notation, we interpret the exclamation point after a field name in an operand list to mean that the parameter to an encoding procedure is a signed integer, which should be narrowed to produce the value of the field.

The branches given with a constructor determine its output pattern. Most commonly, each constructor produces a unique sequence of tokens when applied, so its right-hand side is specified by one pattern containing a single disjunct:

branches ⇒ [`{`*equations* `}`] [`is` *pattern*]

The equations (Section 2.8) are solved to produce a set of *conditions*, which are predicates sufficient to guarantee that the equations have a solution, and to produce *bindings* for the variables solved for. When encoding, the field- and integer-valued operands to the constructor provide the inputs to the equation solver; when decoding, it is the variables bound to fields in *pattern* which perform that function. Equations can be omitted if not needed.

The output pattern, which describes the binary representation of the constructor, can also be omitted, in which case all the field names, pattern names, and constructor-type names in the opcode and operands are conjoined to produce an *implicit* output pattern. Omitting both equations and output pattern yields a very concise constructor specification; we use this idiom heavily in our MIPS and SPARC descriptions.

The toolkit checks a constructor's output pattern for consistency. It generates an error if a field in a token is overconstrained or if fields overlap. For an untyped constructor, it generates a warning if there are unspecified bits in the constructor's output pattern, and it initializes unspecified bits to zero.

Sometimes we want to choose an output pattern based on some properties of the operands (conditional assembly). In that case, we can give multiple branches, each with its own set of equations.

branches ⇒ {`when {` *equations* `} is` *pattern* | `otherwise is` *pattern*}

The sequence of `when` branches may not be empty, and output patterns cannot be omitted when multiple branches are used.

When using this constructor in encoding, the toolkit chooses the first branch whose conditions are known to be satisfied. This conservative approach means that the toolkit can always produce correct code in a single pass, but that, for example, branches to unknown labels are always encoded in the most general way. It would be pleasant if we got around to incorporating standard multi-pass algorithms for conditional assembly, but don't look for it soon.

2.8 Equations and expressions

Equations are written by relating two expressions using the relational operators. Multiple equations are separated by commas.

equations ⇒ *equation* {`,` *equation*}

equation ⇒ *expr relational-operator expr*

Only the `=` operator truly contributes information to a set of equations to be solved. All inequalities in a list of equations are simply added to the list of *conditions* that must be satisfied if the equations are to have a solution.

The expression syntax uses the standard binary operators, unary minus, and special postfix syntax for bit slicing and sign extension:

expr ⇒ *integer*
 | *identifier* [*bit-slice*] [`!`]
 | *expr binary-operator expr*
 | `(` *expr* `)`

binary-operator ⇒ `+` | `-` | `*` | `/`

bit-slice ⇒ `@[` *lo-bit* [`:` *hi-bit*] `]`

The special identifier `_` is a "wildcard." It stands for a "don't care" value, and every instance stands for a different variable.

The binary operators have the standard precedence. Using explicit division is almost certain to cause the solver to complain. Multiplying two unknowns will also break the solver—you should only multiply by integer literals.

A bit slice denotes the range of bits from *lo-bit* to *hi-bit*, inclusive.[9] By default, bit 0 is the least significant bit, but the significance of bit 0 can be

[9] The current syntax for bit slices differs from that used in earlier versions of the toolkit. The old syntax led to parsing conflicts when restrictions on the use of expressions were relaxed.

changed with the `bit 0` directive (Section 2.9.2). If *hi-bit* is omitted, it defaults to *lo-bit*, that is, the slice is one bit wide.[10] Any quantity may be sliced. Only slices and fields may be sign-extended, because they are the only quantities whose widths are known.

The toolkit's equation solver is simple-minded, and it gives terrible error messages.

2.9 Miscellaneous

2.9.1 Addressing units for the program counter

The specification

spec ⇒ `pc_unit_bits` *width*

says that the program counter measures units of *width* bits. For example, if `pc_unit_bits` is 8 (the default), a 16-bit token in a sequence causes the program counter to be advanced by 2. The mechanism of the advance is the second argument to `emit` (when encoding) or the offset passed to the template for address arithmetic (when decoding). We also use these units when converting labels to integers. (We use the difference, in increments of `pc_unit_bits`, between the label and the program counter). The size of any token that appears in a sequence must be exactly divisible by `pc_unit_bits`.

2.9.2 Bit numbering

By default, bits used to specify fields and bit-slices are numbered with bit 0 as the least significant bit. This choice can be awkward for machines that number the bits starting with bit 0 as the most significant bit. The `bit 0` directive sets the bit numbering:

spec ⇒ `bit 0 is` (`most` | `least`) `significant`

This declaration changes the bit numbering used to interpret any `fields` declarations and bit slices that follow it. We recommend against changing bit numberings in mid-specification.

Only the default bit numbering has been tested.

2.9.3 Placeholders

The specification

spec ⇒ `placeholder for` *token-name* `is` *pattern*

Tells the toolkit's generator to use *pattern* when it needs to emit a placeholder for the named token. Placeholders are emitted whenever a relocatable address is needed but not yet known.

[10] It is a defect in the toolkit that there is no way to notate a slice from bit k up to the most significant bit, whatever that may be (e.g., 31 or 64). If you want this defect remedied, suggest a suitable notation.

2.9.4 Word size

There are a few situations in which we use the word size of the machine for which we are generating code—mostly to see whether certain values will fit. It defaults to 32 bits, but it can be specified by

spec ⇒ `wordsize` *width*

where *width* is the width in bits.

2.9.5 Declaring identifiers relocatable

. To write specifications that use relocatable addresses, you must declare which operand names refer to relocatable addresses. Write

spec ⇒ `relocatable` {*identifier*}

to mark one or more identifiers as relocatable addresses. In subsequent constructor definitions, such identifiers denote relocatable addresses when used as operands.

2.9.6 Discarding constructors

You may want to delete constructors from a specification, reducing the number of encoding procedures generated. You may say what to keep or what to throw away.

spec ⇒ `keep` {*opcode*}
  | `discard` {*opcode*}

Constructors (or groups thereof) are named by their opcode specifications.

2.10 Assembly syntax

You can write perfectly interesting encoding and decoding applications without specifying an assembly-language syntax, so you may want to skip this section on first reading and go on to Section 2.11

A constructor's left-hand side, i.e., the constructor's name, its operands, and some syntactic sugar, is often sufficient to specify the constructor's syntax in the assembly language recognized by a vendor's assembler. This is the case for most instructions on the MIPS and SPARC when assembled by Unix assemblers. Other architectures, like the Pentium, are targets of multiple assemblers, each of which may recognize a distinct assembly syntax. Multiple syntaxes prevent the specification writer from using one constructor to specify all variant syntaxes. Moreover, some assemblers overload instruction names, using context to determine which instruction is meant. For example, the Pentium `add` can mean any of five different instructions, depending on the sizes and locations of the operands. The toolkit cannot do this kind of overloading; it must use different names for different instructions (constructors). The reason is that the toolkit must generate a different encoding procedure for each instruction, and in most programming languages, different procedures must have different names.

We solve this problem by requiring each constructor to have a different name. Typically, the specification writer distinguishes variants by adding suffixes to the base name of the constructor. For example, the Pentium `add` instructions include constructors called `addb`, `addib`, `addiowb`, `addmrb`, and `addrmb`. To get from these names back to assembly language, we have to define an appropriate mapping. These mappings, and others for operand syntax, are defined separately from the main specification, because different vendors use different syntaxes for their assembly languages.

An assembly specification includes three parts: a mapping of constructor names to assembly opcodes, the assembly format for each constructor operand, and the assembly syntax for each constructor. We use the Pentium specification of the assembly language supported by the GNU assembler to illustrate assembly specifications. First, we describe assembly name mappings, then operand format and constructor assembly syntax.

spec ⇒ *assembly-opcode-syntax*
  | *assembly-operand-syntax*
  | *assembly-syntax*

2.10.1 Assembly names for opcodes

A constructor's name may contain multiple parts derived from pattern names and constant strings. Each part may or may not contribute to the assembly name. For example, the constructor name `add^"mrb"` contains two parts, the first derived from the pattern `add` and the second from the string `"mrb"`. The assembly name for this constructor is `addb`, so the pattern `add` contributes its name and the suffix `"mrb"` is mapped to the string `"b"`. This example illustrates how the constructor `addmrb` has a more specific name to disambiguate it from other overloaded variants of `addb`.

`assembly opcode` introduces mappings from complete constructor names (strings) to their assembly names (strings). `assembly component` introduces mappings from parts of constructor names to their assembly names.

assembly-opcode-syntax ⇒ `assembly opcode` *glob-pattern* `is` *glob-target*
  | `assembly component` *glob-pattern* `is` *glob-target*

Complete mapping simply enables a specification writer to rename entire constructors. We also provide component-wise mapping because it improves factoring of assembly names among constructors that share common suffixes and prefixes. For example, the suffix `B.Eb.Ib` always maps to `b` in every constructor name where it appears. That mapping is specified by

```
assembly component B.Eb.Ib is b
```

Here's how we disambiguate the two mapping mechanisms. If a complete name mapping exists for a constructor name, it is applied, and componentwise mappings are ignored. If no complete mapping exists, mappings are applied individually to *each* part of a constructor's name and the resulting strings are concatenated into the complete assembly name.

2.10.2 Globbing expressions

There is redundancy in the mapping of constructor names, so we use a regular expression syntax to group related names. The regular expression syntax is the same as the syntax for C-shell "globbing" expressions (Joy 1994). For example, the mappings applied to the parts of the constructor name `add^"mrb"` are:

```
assembly component
    add      is  add
    *b       is  b
```

The first rule maps `add` to itself and the second maps any string that matches `*b` to `b`. The least general rule that matches a string is applied.

What do we mean by "least general?"

It is often useful to define a default mapping, i.e., for the pattern "`*`". On the supported targets, most constructor names map directly to assembly names, so the default maps a name to itself, i.e., `assembly component {*} is $1`.

In globbing expressions, "*" matches any string. Any other character, including the dot, matches itself. The concatenation operator is implicit, so adjacent characters are concatenated. Alternatives are comma-separated lists of strings delimited by "`{`" and "`}`".

The string on the right-hand side may contain elements of the form `$`n, where n is the n-th braced expression on the left-hand side. `$$` stands for a single dollar sign.

$$\textit{glob-pattern} \Rightarrow \left\{\texttt{*} \;\middle|\; \textit{string} \;\middle|\; \textit{glob-alternatives}\right\}$$

$$\textit{glob-alternatives} \Rightarrow \texttt{\{}\ \textit{glob-pattern}\ \left\{\texttt{,}\ \textit{glob-pattern}\right\}\ \texttt{\}}$$

$$\textit{glob-target} \Rightarrow \left\{\textit{string} \;\middle|\; \texttt{\$}\textit{integer} \;\middle|\; \texttt{\$\$}\right\}$$

For example, the rules below specify that the suffixes `ow` and `aw` map to the assembly name `w`, and that `od` and `ad` map to `l`.

```
assembly component
    {o,a}w   is w
    {o,a}d   is l
```

2.10.3 Assembly formats for operands

`assembly operand` introduces mappings from operands to formatted strings that specify how to print the operands in assembly code. Operands may be fields, integer inputs, relocatable addresses, or constant strings. We use `printf`-style syntax for formatted strings.

assembly-operand-syntax ⇒ `assembly operand` *Ident* `is` *operandsyntax*

operandsyntax ⇒ *format-string* `using` *operandname*
| *operandname*

operandname ⇒ `sparse [` *binding* {`,` *binding*} `]`
| `names [` {*Ident* | *String*} `]`
| `field` *Ident*

The first `assembly operand` rule below specifies that immediate operands are prefixed by "$" and are printed as integers. The second rule specifies that the listed field operands are prefixed by "%" and printed as strings, using the names provided in their `fieldinfo` declarations.

```
assembly operand
    [count i8 i16 i32]           is "$%d"
    [r32 sr16 r16 r8 base index] is "%%%s"
```

Some inputs are not declared as fields but should be printed in the same format as fields. For example, the input `reg` should be printed using the names associated with the field `base`. The following rule uses the optional `using` *field* clause to specify that `reg`, `reg8`, etc. should be printed using the fieldinfo associated with `base`.

```
assembly operand
    [reg reg8 sreg cr dr]       is "%%%s" using field base
```

2.10.4 Assembly syntax for constructors

The default assembly syntax for a constructor appears in the constructor's specification; an alternate syntax may be specified with `assembly syntax`. Providing assembly syntax in a constructor specification can help a specification writer or user read and identify a constructor, and it is concise when only one assembly syntax is required. An alternate syntax may be needed, however, if more than one assembly language is used on the target.

`assembly syntax` uses the same syntax as the `constructors` directive: a constructor name followed by a list of operands.

assembly-syntax ⇒ `assembly syntax` *opcode operands*

The assembly-syntax specification must use the same set of operands that the constructor uses, but the operands may appear in any order and with any syntactic sugar. For example, the GNU assembly language reverses the order of some constructor's operands, such as `arithI^"d"` below.

```
assembly syntax
  arithI^"d"     i32!, Eaddr
  MOV.Eb.Iv^od   i32!, Eaddr
```

Some constructors have implicit operands, such as the floating-point stack constructors. Assemblers require those operands to distinguish between overloaded floating point instructions.

```
assembly syntax
  fstack^Sstack  "%st", "%st"(idx)
```

The GNU syntax for effective addresses bears no resemblance to the DOS syntax, but it can be specified in only a few lines.

```
assembly syntax
  Indir       (reg)
  Disp32      d(reg)
  Index       (base,index,ss)
  Index32     d(base,index,ss)
  ShortIndex  d(,index,ss)
```

2.11 Lexical details

The lexical structure used in specifications is similar to that of C, except that dots as well as underscores are treated as letters for purposes of finding identifiers. In particular, all names are case-sensitive; for example, `Address` is not the same as `address`. Strings are delimited with double quotes, but the C "backslash escapes" aren't used—instead, a double quote within a string is represented by two consecutive double quotes, as in Pascal. The toolkit supports the C-style `0x` and "leading zero" notations used to give integer literals in hexadecimal or octal bases. It also provides a `0b` notation, analogous to `0x`, for specifying integer literals in binary. Integer literals may also be specified as C-style character constants. For reasons that make no sense whatever, these constants *do* use the C "backslash escapes."

The toolkit reserves the following words as keywords:

```
...         constructors  fields   otherwise     syntax
_           discard       for      patterns      to
address     else          is       pc_unit_bits  type
any         endmatch      keep     placeholder   using
assembly    epsilon       names    relocatable   when
bit         fetch         of       some          which
columns     field         opcode   sparse        wordsize
component   fieldinfo     operand
```

These words cannot be used to name constructors, fields, patterns, etc.

Comments in SLED specifications begin with `#` and go to the end of the line. Comments in matching statements are the comments of the target programming language, and they may appear only where target-language code may appear.

Chapter 3

Decoding applications and matching statements

Decoding applications are written using ordinary C or Modula-3 with embedded *matching statements*. The matching statements are like case statements, except that the arms are labelled with patterns, not with values. The toolkit's translator transforms the matching statement into ordinary code, which is then compiled as usual. Supposing that the file `decode.m` is part of a decoding application and contains one or more matching statements, this transformation can be specified in a Makefile as follows:

```
decode.c: decode.m
        tools -decoder decode.c -matcher decode.m m.spec i.spec
```

where `m.spec` contains a machine description and `i.spec` contains instruction-stream templates (see Section 3.2). Typically the `m.spec` file can be shared among a variety of applications, but different applications may use different representations of instruction streams, and therefore different `i.spec` files.

The toolkit works by computing a *decision tree* for each matching statement. Each internal node of the tree tests a particular field of a token from the input stream. No field is tested more than once, and no token is fetched more than once. The leaves of the tree contain single arms of the matching statement.[1] At each leaf, the toolkit solves equations to compute the values of free variables in the matching pattern; the inputs to these equations are the values of the fields tested. The toolkit turns the tree into a dag by combining identical leaves; this process can eliminate unnecessary tests. Finally, the toolkit generates C or Modula-3 code from the dag.

The rest of this chapter explains the syntax of matching statements and the way to specify the representation of instruction streams using templates.

[1] If two arms match the same pattern, but with different equations, they are both assigned to the same leaf, and the equations have to be tested to distinguish them.

3.1 Matching statements

The toolkit's matching statements are inspired by pattern matching in Standard ML (Milner, Tofte, and Harper 1990). Matching statements are embedded directly in C or Modula-3 code. A matching statement begins with

`match` *code* `to`

on a line by itself. The *code* is an expression that computes a value of type "address," as specified by the `address type` template. This address points into an instruction stream at the point where decoding is done. The body of a matching statement contains a series of arms, and the matching statement ends with `endmatch` on a line by itself.

The first line of an arm begins with a | and has the following syntax:

| *pattern* [`{` *equations* `}`] [`[` *name* `]`] `=>` *code*

Free variables of *pattern* or *equations* are available to be used in the *code* on the right-hand side. Free variables from *pattern* can come from field constraints or from constructor applications; in either case, all such variables represent binding instances. If a free variables is an integer-typed argument to a constructor or it appears in field bindings, it is an integer. If a free variable is a constructor-typed argument to a constructor, then on the right-hand side of the `=>` it stands for the location in the instruction stream where that argument matches. It's as if the name of the argument were written as a label preceding each constructor of the appropriate type. The location is converted to an integer by the usual mechanism (arithmetic from the matched address in units of `pc_unit_bits`).

The special variable `_` can be used in constructor applications to stand for operands whose values are uninteresting. The optional *name*, if present, is an identifier that is bound to the name of the *pattern*. This name may be a name introduced by a `patterns` declaration, or it may be the name of a constructor that was applied to produce the pattern. (Patterns lose their names when combined with other patterns using the pattern operators.) The *code* may occupy multiple lines; the toolkit takes all the text between the `=>` and the next initial |, or the end of the matching statement.

The last arm of a matching statement may be written

`else` *code*

which is syntactic sugar for

| `epsilon =>` *code*

Wildly abusing notation, we could summarize the matching statement as follows:

`match` *code* `to`
{| *pattern* [`{`*equations* `}`] [`[` *name* `]`] `=>` *code*}
[`else` *code*]
`endmatch`

That's not quite the whole story, however. It's common to want to know how many tokens are consumed by a matching statement. One can do this by adding, e.g., `; tail: epsilon` to every pattern, and by saving `tail` in every arm, but the bookkeeping gets old quickly. It's possible instead to put an identifier in square brackets immediately after `match`; the toolkit will assign the address of the first unread token to that identifier. The syntax is therefore

> `match` $\big[$`[` *var-name* `]`$\big]$ *code* `to`
> $\big\{$ `|` *pattern* $\big[$`{`*equations* `}`$\big]$ $\big[$`[` *name* `]`$\big]$ `=>` *code*$\big\}$
> $\big[$`else` *code*$\big]$
> `endmatch`

var-name should be declared as a variable of address type (*outside* the matching statement).

The decoder generated by the toolkit has the effect of checking each arm of the matching statement in turn. As soon as it finds an arm whose pattern matches and whose equations have a solution,[2] it binds all the free identifiers on the left-hand side of the arm, then executes the code on the right-hand side. The decoders the toolkit generates are actually much more efficient, testing inputs against all arms at once, but the sequential model accurately describes the semantics.

3.2 Access to instruction streams during decoding

Decoders generated by the toolkit's translator instantiate code templates to get access to the tokens of the instruction stream being decoded. The toolkit treats addresses and instruction streams as abstract data types; these templates give the operations.

> *spec* ⇒ `fetch` (*width* | `any`) `using` *template*
> | `address type is` *template*
> | `address add using` *template*
> | `address to integer using` *template*

Each *template* is a string. `fetch` templates are used to fetch words from an instruction stream. Some applications may be able to give a single `fetch` templates that works with any width; others may have to give a different `fetch` template for each width used in the specification. `address type` identifies the type the application uses to represent a location in an instruction stream; the `address add` template is used for address arithmetic, which moves through the stream. `address to integer` is used to convert an address to an integer; it is used to assign values to labels. In the templates, `%a` stands for an

[2]Note that patterns obtained by constructor application may carry along some equations implicitly.

address, `%o` for an offset, and `%w` for a width. Offsets are measured in increments of `pc_unit_bits`, but widths are measured in bits. `fetch` templates may use `%a` and `%w`. The `address add` template may use `%a` and `%o`. The `address to integer` template may use only `%a`. The `address type` template doesn't use any escapes.

Judicious definitions of the templates can help handle standard object files, in which the address of an instruction in memory need not correspond to the address that instruction would occupy in a running program. For example, if the pointer `instr` pointer points to a buffer containing instructions, and the integer `pc` indicates the address that buffer would occupy in a running program, the following templates may prove helpful:[3]

```
address type is "unsigned char *"
address fetch 8 using "*(%a)"
address add using "%a+%o"
address to integer using "(%a-instr)+pc"
```

These templates would be suitable for use with a matching statement beginning, e.g., `match instr to`. This transformation requires care; matching on an address (e.g. for addressing modes) produces an answer in "integer-space," where the beginning of the block is at `pc`. To use that answer in another matching statement, it would have to be transformed back into "address-space," where the beginning of the block is at `instr`.

3.3 Debugging generated decoders

It can be tricky to understand the structure of a decoder by looking at the generated code. The `-ascii-dag` and `-ascii-tree` options dump complete listings of decision dags and trees, but it can be difficult to understand these dumps. The toolkit can also generate Icon code that you can use to explore trees, viewing one path at a time. This Icon code, when combined with a suitable main program, can act as a CGI script on the World-Wide Web, and you can explore the trees using your favorite Web browser. This section shows how to set up that facility.

Let's suppose you have a SLED specification called `problem.spec`, and you've written one or more matching statements in the file `problem.m`.

1. Create the Icon file used to describe the matching statements:

   ```
   tools -icon-dag problem.icn -matcher problem.m problem.spec
   ```

   The writes a description of the generated matchers into `problem.icn`.

2. Combine this Icon file with the `htmltree.icn` file supplied with the toolkit distribution, and call the resulting program `problem`:

[3] This code has not been tested.

```
icont -o problem problem.icn htmltree.icn
```

3. Put the resulting executable where it can be executed as a CGI script. You'll probably need help from your friendly local systems administrator. Here's what I had to do at my site, which runs the Apache 1.0.3 server; your mileage may vary:

   (a) Put the executable Icon program in the directory `/home/nr/public_html/cgi-bin/trees/`. The corresponding URL is `http://www.cs.virginia.edu/cgi-bin/cgiwrap/nr/trees/`.

   (b) The code in `htmltree.icn` uses the environment variables `SERVER_NAME` and `SCRIPT_NAME` to reconstruct the URL. With luck, these will be independent of server. Ha ha.

   (c) For some unknown reason, Icon programs won't run as CGI scripts at my site, so I had to wrap things in this shell script:

   ```
   #!/bin/sh
   tree="$1"
   shift
   SCRIPT_NAME="${SCRIPT_NAME}?$tree"
   export SCRIPT_NAME
   exec /home/nr/public_html/cgi-bin/trees/$tree "$@"
   ```

4. Use the URL to explore the matching statements. You can see matcher for the sample SPARC disassembler at `http://www.cs.virginia.edu/cgi-bin/cgiwrap/nr/trees/view?sparcdis`.

Chapter 4

Encoding applications and the toolkit's library

Encoding applications generate instructions in binary form by calling two kinds of toolkit procedures: machine-dependent encoding procedures that are generated from a machine description, and machine-independent procedures that are part of the toolkit's library. These procedures require some support from the application, for example, to allocate space for the relocatable blocks into which instructions are emitted. This support is explained in Section 4.7.

Since version 0.2, the toolkit has also been able to generate encoding procedures that emit assembly language. These procedures require additional support from the application, e.g., a method of printing.

This chapter specifies the library functions that are used by the toolkit-generated encoding procedures and by the application program. The library includes support for instruction and data streams, in the form of relocatable blocks, and for reading and writing the streams. If the application writer chooses to use his own implementation of streams, the implementation must conform to the specifications given here.

The library is written in C, we specify it by writing ANSI C prototypes for its procedures. Some of the library "procedures" are actually implemented as macros; for such macros, we provide prototypes that describe the equivalent procedures.

4.1 Varieties of encoding procedures

The toolkit can generate two kinds of encoding procedures for untyped constructors: one emits the binary representation of instructions and the other emits the assembly representation. By default, binary encoding procedures call the procedure `emitm`, which emits tokens in the byte order of the host machine into the current instruction stream. The toolkit's library provides an implementation of `emitm`, as well as `emitb`, which emit tokens in big-endian order, and `emitl`, which emits in little-endian. A user can provide alternative implementations. At toolkit execution time, the `-byteorder X` option specifies an alternate emitter function named `emitX`; `l` and `b` are reasonable alternatives for `X`. The user can delay choosing an emitter function until compile time, by compiling the binary encoding procedures with `cc -DemitX=`*emitter-name*. This trick avoids running the generator multiple times, which can be expensive.

Assembly encoding procedures call the `asmprintf` procedure to emit assembly code onto `asmprintfd`. They call `asmprintreloc` to print relocatable addresses. The library initializes `asmprintfd` to standard output and `asmprintf` to `fprintf` (or an equivalent procedure). `asmprintreloc` prints relocatable addresses in the format *label-name+offset*, if the address' label is defined, otherwise it prints `???+`*`offset`*. Applications can choose alternative print procedures by overriding these values.

Unless a field is specified as `unchecked`, both binary and assembly encoding procedures check that the values of the field's operands are in its range. If unchecked, they do not check an operand's value, but they do mask out the operand's high bits, ensuring that the token is not corrupted by a bad value. If a field is specified as `guaranteed`, the encoding procedure does not check or mask the operand's value.

Encoding procedures are declared external by default and therefore, their names are in the global-name space of the application program that uses them. Often this is inconvenient: encoding procedures' names may conflict with procedure names in the application or the application may need to invoke dynamically multiple sets of encoding procedures, e.g., both binary and assembly. To avoid these problems, the user can invoke the toolkit with `-indirect name[:type]` option, which isolates encoding procedures in a single compilation unit and provides indirect access to them through a structure of procedure pointers. `name` gives the name of the structure and `type` is its type; if `type` is omitted, we use `name`. This permits an application, like the toolkit's checker, to use multiple sets of encoding procedures and to switch between them dynamically. For example, the toolkit checker for the MIPS includes two structures: `mips_asm` for assembly encoding procedures and `mips_bin` for binary. They are generated by using `-indirect mips_asm` and `-indirect mips_bin:mips_asm`; the latter option guarantees that both structures have the same type and therefore, can be used interchangeably.

4.2 Building an encoding application

The toolkit generator transforms a machine description `m.spec` into encoding procedures and their interfaces. The interface and implementation are written to different files. The following rules for a Makefile specify how to create and use encoding procedures. The first rule emits encoding procedures into the file `encoder.c` and the interface into file `encoder.h`:

```
encoder.c encoder.h: m.spec
	tools -encoder encoder m.spec
```

Application source that includes calls to the encoding procedures must include the encoding procedures' header file as well as `mclib.h`, the header file for the toolkit library.

```
app.o: app.c encoder.h mclib.h
        cc -c app.c
```

Users may have to add an `-I` option so the C preprocessor can find `mclib.h`.

The application executable must include the encoding procedures and the library itself. The last three rules compile the encoding procedures and library source and link the complete application.

```
encoder.o: encoder.c encoder.h mclib.h
        cc -c encoder.c
mclib.o: mclib.c mclib.h
        cc -c mclib.c
app: app.o encoder.o mclib.o
        cc  app.o encoder.o mclib.o -o app
```

When compiling `mclib.c`, compilers for 32-bit hosts may issue warnings about a "shift count larger than the size of the type." These warnings are to be expected and should be ignored.

The rest of this chapter describes the types, functions, and macros that appear in `mclib.h`.

4.3 Labels and relocatable blocks

A *relocatable block* is the smallest unit of information relocatable by the library and is used to represent a stream of instructions or data. Relocatable blocks can represent information of various types. A traditional linker, for example, might associate one relocatable block with each section in an object file. An incremental linker might associate one relocatable block with each procedure, which would facilitate relocation of individual procedures at link time.

A relocatable block `rb` has the following fields:

`rb.lc`	the current location counter (offset) in `rb`.
`rb.size`	the size of the relocatable block.
`rb.low`	the lowest value of `rb.lc` for which the contents of `rb` are known, or null if nothing is known of the contents of `rb`.
`rb.address`	the absolute address of `rb` if that address is known, otherwise null.
`rb.contents`	a buffer storing the contents of `rb`. Clients may refer to the elements `0..rb.size-1` of `rb.contents`. The values of `rb.contents[0..rb.low-1]` are undefined, and it is illegal to attempt to write these values. (This restriction enables an implementation that handles completely uninitialized blocks without allocating any memory.)
`rb.label`	an unnamed label that always refers to the location of the beginning of (`rb.contents[0]`).

Because of performance optimizations within the library, it is not safe for applications to read or write these fields directly. Most fields may be read using one of these macros:

⟨*library types*⟩≡
```
typedef struct reloc_block *RBlock;
```
⟨*faux declarations of library macros*⟩≡
```
extern unsigned block_lc(RBlock rb);
extern unsigned block_size(RBlock rb);
extern unsigned block_low(RBlock rb);
extern int      block_address_known(RBlock);
extern unsigned block_address(RBlock);
extern RLabel   block_label(RBlock);
```

The `contents` aren't directly accessible as values; instead, we provide procedures to copy them into a buffer or a file (see Section 4.8). Except for the `address` field, this interface provides no way to tell whether fields are `null`. These macros return implementation-dependent values instead of `null`.

The fields of a relocatable block can be written using the procedures defined below.

A *label* is a location in a relocatable block. All undefined labels are located in a distinguished relocatable block, `undef`, whose `address` field is always null. A label `l` has the following fields:

`l.block`	the relocatable block in which `l` is located
`l.offset`	`l`'s offset from the beginning of `l.block`
`l.name`	a string associated with `l`

The name field may be used by the application to associate labels with names of important objects, e.g., procedures and global symbols.

⟨*library types*⟩+≡
```
typedef struct label *RLabel;
```

4.3.1 Operations on relocatable blocks

`block_new` creates a new relocatable block, `rb`, with `rb.lc = rb.size = 0`, null `rb.address` and `rb.low`, and empty `rb.contents`. If the argument `size` passed to `block_new` is non-zero, the library uses it as a hint about the maximum size the relocatable block is expected to reach. If the hint is accurate, the library will make more efficient use of memory. `block_new` adds each newly created block to `defined_rbs`, the set of all relocatable blocks created, excluding `undef`. `block_defined` succeeds if the given block is defined, i.e. it is not `undef`.

⟨*library procedures*⟩≡
```
extern RBlock block_new(unsigned size);
```
⟨*faux declarations of library macros*⟩+≡
```
extern int block_defined(RBlock rb);
```

`set_lc` sets the value of `rb.lc` and also updates `rb.size` if it is lower than `rb.lc`; `rb.size` is the "high-water mark" of `rb.lc`.

⟨*library procedures*⟩+≡
```
extern void set_lc(RBlock rb, unsigned lc);
```

In order to accomodate traditional assembly languages, the library maintains a current relocatable block `crb`. Many of the procedures in this interface have side effects on `crb`. Initially, `crb` is undefined. `set_block` sets the current relocatable block to `rb` and `crb` produces the current relocatable block.

⟨*library procedures*⟩+≡
```
extern void set_block(RBlock rb);
```
⟨*faux declarations of library macros*⟩+≡
```
extern RBlock crb(void);
```

Relocation occurs when an absolute address is assigned to a relocatable block. `set_address` sets the address of a block. It can be called multiple times to to change the address.[1] `block_address` produces the address. Its value is meaningful if and only if `block_address_known` returns nonzero; otherwise the address is null.

⟨*library procedures*⟩+≡
```
extern void set_address(RBlock, unsigned);
```
⟨*faux declarations of library macros*⟩+≡
```
extern int      block_address_known(RBlock);
extern unsigned block_address      (RBlock);
```

An application may want to reserve a register to point into a relocatable block. For example, the MIPS short-data (`.sdata`) section could be treated as a relocatable block, in which case the reserved register is the global pointer

[1] For efficiency, some applications may want to discard relocation information associated with a relocatable block `rb`. That would require the ability to mark `address` read-only. Let us know if you are interested.

(gp). If an application wants to reserve more than one register, it should split the relocatable block into multiple blocks. The following two fields make it possible to reserve registers:

`rb.reg`	if non-null, it identifies the register that points to `rb.contents[rb.reglc]`.
`rb.reglc`	the offset within the block to which `rb.reg` points.

The application notifies the library that a register points into a relocatable block by calling `set_register`. The application or toolkit can query the value of the register or its absolute address by calling `block_reg` or `block_regaddr`. Note that the value of `reg` is undefined if `set_register` has not been called; `regaddr` is similarly undefined if `reg` or `address` is undefined. It might be desirable to have some error checking to avoid binding two different registers to the same relocatable block; an `unset_register` procedure might help but we don't have one right now.

⟨*library procedures*⟩+≡

```
extern void set_register(RBlock rb, unsigned reg, unsigned regaddr);
extern unsigned block_reg(RBlock rb);
extern unsigned block_regaddr(RBlock rb);
```

4.3.2 Operations on labels

`label_new` creates a new label located in `undef` and assigns `name` to `l.name`. `label_define` binds a label to the given `offset` in the current relocatable block. Once a label is defined, it can't be redefined.[2] `label_define_at` lets you name the block and offset explicitly.

⟨*library procedures*⟩+≡

```
extern RLabel label_new(char *name);
extern void label_define(RLabel lbl, int offset);
extern void label_define_at(RLabel lbl, RBlock block, unsigned lc);
```

Because `label_define` uses an offset, the first pass of an assembler can maintain its own location counter in a register, using that counter to define labels. This trick means one library call per label instead of one per instruction.

The absolute address of a label is returned by `label_location`. Its value is defined only if `label_location_known` is true.

⟨*faux declarations of library macros*⟩+≡

```
extern int label_location_known(RLabel lbl);
extern unsigned label_location (RLabel lbl);
```

[2]For most applications (e.g., an assembler, `mld`), it is wrong to redefine a label, but an incremental linker might want to redefine a label when a relocatable block is replaced. We're willing to reconsider this decision.

4.4 Emitting information

The library is set up to make it easy and efficient to emit instructions or data at the current location in the current relocatable block. Such emission automatically advances the current relocatable block's location counter (`crb.lc`).

Most applications emit information one word at a time, not one byte at a time, raising the spectre of byte order. Byte order could be made part of the global state of the library, or it could be selected by supplying a parameter to an emitter procedure, but we have chosen to provide different emitter procedures for different byte orders. Applications that use only one byte order enjoy maximum efficiency; others can get reasonable efficiency by using procedure variables.

We provide two procedures that use fixed byte orders, plus a pointer to a third procedure that detects the byte order of the host machine and selects the matching procedure. `*emitm`, `emitl`, and `emitb` convert the least significant 8`n` bits of `value` to a string of bytes, using the endianness requested (`m`achine, `l`ittle, or `b`ig). The `contents` of the current relocatable block is extended if it is not big enough to hold the new bits, and the bits are written to that `contents` starting at the current location. The location counter is incremented by `n`.

⟨*library procedures*⟩+≡
```
extern void (*emitm)(unsigned long value, unsigned n);
extern void emitl(unsigned long value, unsigned n);
extern void emitb(unsigned long value, unsigned n);
```

Emitting bytes sets the low-water mark of the current relocatable block if it isn't set already. It's illegal to emit bytes below the low-water mark.

Generated encoding procedures call one of these procedures to emit instructions, as selected by the generator's `-byteorder` option.

In some applications, the overhead required by the emitting procedures can be significant. This overhead includes not only the cost of a procedure call, but also the costs of manipulating bytes in a well-defined order. For example, it is in general unsafe to use a full-word store to place a full word into a relocatable block, even when the target byte order matches that of the host machine, because the alignment may not be correct. Some applications (e.g., RISC code generators) always maintain proper alignment. Such applications can achieve significant speedups by using the `emitfast` macro, which has the same effect as `emitm`, provided that the location counter is properly aligned to the size of the token being emitted. When the location counter is not properly aligned, the effect of `emitfast` is undefined, so it is in general unsafe.

⟨*faux declarations of library macros*⟩+≡
```
extern void emitfast(unsigned long value, unsigned n);
```

For convenience, we provide macros that emit full and half words and individual bytes using the byte order of the machine. They are simply abbreviations for calls to `emitm`.

⟨*faux declarations of library macros*⟩+≡
```
extern void emitu(unsigned value);
extern void emitus(unsigned short value);
extern void emituc(unsigned char value);
```

It would also be convenient to have a procedure for emitting relocatable addresses, but that one would be hard to write. Luckily, users can generate it from a specification like this one:

⟨*specification for generation* `emit_raddr`⟩≡
```
fields of addrword (32) addr32 0:31
placeholder for addrword is addr32 = 0xbadbad
constructors emit_raddr reloc is addr32 = reloc
```

4.5 Manipulating the location counter

The current location counter can be changed by `align`, `org`, and `addlc`. `align` aligns the current location counter on an `n`-byte boundary, `org` sets the location counter to `lc`, and and `addlc` adds `n` bytes to the location counter's current value.

⟨*library procedures*⟩+≡
```
extern void align(unsigned n);
```
⟨*faux declarations of library macros*⟩+≡
```
extern void org(unsigned lc);
extern void addlc(unsigned n);
```

Applications may also change the location counter of any relocatable block using `set_lc`, as described above.

The application can read `crb`'s location counter using `lc`.

⟨*faux declarations of library macros*⟩+≡
```
extern unsigned lc(void);
```

The toolkit is optimized for advancing location counters, so it is more efficient to use `addlc(n)` than to use `set_lc(crb(), lc()+n)`, because `set_lc` must handle the general case.

4.6 Relocation and closures

Computation involving relocatable addresses may be delayed until the location of a relocatable block is fixed. For example, the `emit_raddr` constructor specified above emits a 32-bit relocatable address into the current relocatable block. If `reloc` is unknown when `emit_raddr` is applied, the toolkit generator emits the place-holder pattern `addr32 = 0xbadbad` and creates a *relocation closure* (`RClosure`). In general, the toolkit generator emits a place-holder and creates a relocation closure for any instruction that refers to a relocatable address whose value is unknown. A closure captures the information necessary for delayed evaluation of expressions involving relocatable addresses. When a relocation closure is applied, it overwrites the place-holder instruction with the relocated instruction.

Each relocation closure writes a sequence of tokens starting at a single location in a single relocatable block. Every closure uses, or *depends on*, at least one relocatable address. A relocation closure `cl` has the following fields:

`cl.dest_block`	the relocatable block to be written by `cl`
`cl.dest_lc`	the location written
`cl.depends_on`	the set of relocatable addresses on which the closure depends
`cl.apply`	a function that writes to `cl.dest_lc`, using the values of all the addresses in `cl.depends_on`

Applications may apply closures individually, as the relocatable addresses on which they depend become known, or all at once, after all relocatable blocks have known addresses. Because time of relocation is application-dependent, the library relies upon the application to save relocation closures in whatever manner it finds most convenient and then to apply the closures when ready. The toolkit requires the application to implement `mc_alloc_closure`, which allocates space for a closure and saves it in an application-dependent data structure; see Section 4.7. The library provides `apply_closure`, a closure-independent procedure that applies the closure `cl`.

⟨*library types*⟩+≡
```
typedef struct rclosure *RClosure;
typedef void (*Emitter)
    (RBlock, unsigned dest_lc, unsigned long bits, unsigned size);
typedef void (*FailCont)(char *fmt, ...);
```
⟨*library procedures*⟩+≡
```
extern void apply_closure(RClosure cl, Emitter emitter, FailCont failc);
```

If any of the addresses on which `cl` depends is unknown, the failure continuation `failc` is applied. It should not return (see Section 4.7). The `Emitter` procedure performs the same functions as the `emit` procedures specified above, but instead of writing the current relocatable block, it updates the relocatable block `dest_block` at the location `dest_lc`. The library provides `Emitter` procedures in three byte orders: machine order, little-endian, and big-endian.

⟨*library procedures*⟩+≡
```
extern void (*cl_emitm)(RBlock dest_block, unsigned dest_lc,
                          unsigned long bits, unsigned size);
extern void cl_emitl (RBlock dest_block, unsigned dest_lc,
                          unsigned long bits, unsigned size);
extern void cl_emitb (RBlock dest_block, unsigned dest_lc,
                          unsigned long bits, unsigned size);
```

If the application writer provides an `Emitter` procedure to `apply_closure`, its action must not depend on the values of the current relocatable block or current location counter used by the library's `emit` procedures.

4.7 Application's obligations

The library relies on the application to allocate and store relocation closures, apply closures once the relocatable addresses on which they depend are known, and provide memory-management services to the library, which needs to allocate relocatable blocks, labels, relocatable addresses (fixed size) and contents of blocks (variable size).

The application must provide an allocator named `mc_alloc`, which accepts two parameters. The first is the size of the object to be allocated, and the second indicates what is being allocated. The second parameter may be used by the application to allocate objects from different memory pools; it tells the application whether memory is being allocated for a relocatable block, the contents of a relocatable block, a label, or a relocatable address.

⟨*library types*⟩+≡
```
typedef enum mc_alloc_pool {
    RBlock_pool=1, RBlock_data_pool, RLabel_pool, RAddr_pool
} Mc_alloc_pool;
```
⟨*declarations of procedures provided by the application*⟩≡
```
extern void *mc_alloc(int size, Mc_alloc_pool pool);
```

Applications must also provide `mc_alloc_closure` procedure to allocate memory for a new closure. We use a separate procedure because applications might want to use the destination block and location as hints to determine from what memory to allocate the closure:

⟨*declarations of procedures provided by the application*⟩+≡
```
extern RClosure mc_alloc_closure(unsigned size,
                                 RBlock dest_block, unsigned dest_lc);
```

`mc_alloc_closure` is called by generated encoding procedures, which fill in the fields of the new closure.

Generated encoding procedures call `*fail` if they are called upon to do something they can't, i.e., if their equations can't be solved. For example, the MIPS encoding procedures call `*fail` if asked to branch to an address that is not aligned to a 4-byte boundary. Applications must initialize `fail` to point to a suitable procedure (e.g., one that prints an error message, and maybe exits or `longjmp`s).

⟨*exported data declarations*⟩≡
```
extern void (*fail)(char *fmt, ...);
```

`*fail` may be used as a failure continuation when calling `apply_closure`.

The application is not obligated to provide code used to print assembly language, but it can override the toolkit's choice if it wants to. See Section 4.11.

4.8 Accessing contents of relocatable blocks

To access data already emitted in a relocatable block, applications may call `block_write` or `block_copy`. `block_write` writes the `n` bytes of `rb`'s `contents` beginning at offset `start` to the file descriptor `fd`; `block_copy` is similar but copies into the buffer `buf` instead of writing to `fd`; the application must allocate memory for `buf` before calling `block_copy`.

⟨*library procedures*⟩+≡
```
extern void block_write(int fd,RBlock rb,unsigned start,unsigned n);
extern void block_copy(unsigned char *buf,
                       RBlock rb, unsigned start, unsigned n);
```

We also provide individual access to the contents of relocatable blocks, in three byte orders: machine order, little-endian, and big-endian.

⟨*library procedures*⟩+≡
```
extern unsigned long (*block_fetchm)(RBlock dest_block,
                                     unsigned dest_lc, unsigned size);
extern unsigned long   block_fetchl (RBlock dest_block,
                                     unsigned dest_lc, unsigned size);
extern unsigned long   block_fetchb (RBlock dest_block,
                                     unsigned dest_lc, unsigned size);
```

4.9 Relocatable Addresses

Encoding applications often want to work with addresses whose values aren't known. Common examples include calls to unknown procedures, or forward branches within a procedures. This kind of work can be done with labels alone, but it is more convenient to use a more general abstraction: the *relocatable address*. In its most general form, a relocatable address is an expression whose value depends on one or more labels.[3] Most of the toolkit treats relocatable addresses simply as an abstract data type:

⟨*library types*⟩≡
```
typedef struct relocatable_address *RAddr;
```

The operations available on a relocatable address are to see whether it has a known value, and to extract that value:

⟨*faux declarations of library macros*⟩≡
```
extern int location_known(RAddr addr);
extern unsigned location (RAddr addr);
```

[3]Note that our definition is different from the more traditional definition, in which, for example, the difference of two labels is taken to be an absolute address, not a reloctable address. This traditional definition works only if the two labels are in the same compilation unit.

`location_known(addr)` will be true whenever all the labels on which `addr` depends have known values. If an application writer wants to change our representation of relocatable addresses, his only obligation is to provide `location_known` and `location`.

It is often convenient to refer to the "program counter" (`pc`), i.e., the address of the current location in the current relocatable block, as if it were a relocatable address. In our current scheme, it is not representable as a relocatable address, but we provide a similar interface through macros:

⟨*faux declarations of library macros*⟩+≡
```
extern int cur_pc_known(void);
extern unsigned cur_pc(void);
```

`cur_pc` returns `pc`'s value if `cur_pc_known` is true.

The representation we have chosen for relocatable addresses does not support the fully general form defined above, because we would need a little language for representing and interpreting addressing expressions, and we despair of devising a language whose definition and implementation would be acceptable to all application writers. Instead, the toolkit provides relocatable addresses of the form $L + n$, where L is a label and n is an integer offset. A relocatable address `ra` therefore has two fields:

```
ra.label
ra.offset
```

and one can be created by calling the following procedure:

⟨*library procedures*⟩≡
```
extern RAddr addr_new(RLabel label, int offset);
```

You can also convert an unsigned integer to a relocatable address:

⟨*library procedures*⟩+≡
```
extern RAddr unsigned_to_raddr(unsigned lc);
```

4.10 Excerpts from `mld`, an encoding application

`mld` is an optimizing, retargetable linker that links an intermediate code before generating object code for a target architecture. It performs the functions of a conventional assembler, i.e., it generates object code, and of a conventional linker, i.e., it assigns absolute addresses to data and text segments, places data and procedures in the appropriate segments, and relocates references to external symbols. In this section, we provide some snippets of code from `mld` to illustrate how the library is used in an encoding application. Many of the examples are target independent and are used by `mld` for each target architecture. The target-dependent excerpts are for the MIPS.

4.10.1 Using encoding procedures

`mld`'s code generators are based on those used in the `lcc` compiler, which emit assembly code (Fraser and Hanson 1991). Much of each code generator is generated from a BURG specification (Fraser, Henry, and Proebsting 1992), which contains rules for rewriting intermediate-code subtrees to assembly-language templates; the rest of the code generator is written by hand. Adapting a code generator means modifying both the BURG specification and the hand-written parts. The toolkit simplifies those modifications. Code that prints assembly language is replaced with code that calls the encoding procedures generated by the toolkit; the following three rules are extracted from `mld`'s set of rules that map assembly-language templates to calls to encoding procedures. The format arguments `%c` and `%0` refer to the registers chosen by the code generator.

```
"cvt.d.s $f%c,$f%0"      cvt_d_s(%c, %0), nop()
"mov.d $f%c,$f%0"        mov_d(%c, %0)
"not $%c,$%0"            nor(%c, 0, %0)
```

Hand-written assembly statements are replaced in a similar manner; for example,

```
print("fadds %f1, %f2, %f3");
```

is replaced by

```
fadds(f1, f2, f3);
```

a call to a toolkit-generated encoding procedure. The rules for mapping assembly to binary must be written for each target architecture.

4.10.2 Using relocatable blocks

The encoding procedures use the toolkit's library to write the generated code into relocatable blocks in `mld`'s memory; `mld` calls the library to create relocatable blocks and to set the current relocatable block. At startup, `mld` assigns one relocatable block to each data section by calling `block_new`. `mld` uses four of the MIPS data sections: read only (`rdata`), short data (`sdata`), initialized data (`data`) and uninitialized data (`bss`). `mld` pre-computes the sizes of each data section and provides these sizes to `block_new` as hints of the maximum size in bytes of each relocatable block. The following snippet initializes the blocks.

⟨`mld` *example*⟩≡

```
rdata = block_new(rdata_size);
sdata = block_new(sdata_size);
data  = block_new(data_size);
bss   = block_new(bss_size);
```

`mld` assigns a relocatable block to each procedure for which it generates object code by calling target-independent `new_procedure`. In `new_procedure`, `text` is a global variable that refers to the relocatable block of the current procedure. `set_block` is called first to set the current relocatable block to the current procedure. `block_new` is then called to create a new relocatable block; it uses `size` as a hint of the procedure's maximum size in bytes. The procedure's absolute address in the text segment is set using the current `pc` as its start address; `align(4)` aligns the `pc` on a 4-byte word [4] If the current `pc` is unknown, the base address of the code segment is used. Finally, `set_block` sets the current relocatable block to the new text block. All subsequent calls to encoding procedures will emit instructions in the new relocatable block.

⟨mld *example*⟩+≡
```
static void new_procedure(size, name) char *name; {
    set_block(text);
    text = block_new(size);
    text->label->name = name;
    align(4);
    set_address(text, cur_pc_known() ?
                      cur_pc() : segments[CODE].baseaddr);
    set_block(text);
}
```

The adapted code generators also call the library directly, e.g., to emit data. The target-dependent `segment` procedure sets the segment in which data is emitted; the current relocatable block is set to the appropriate segment.

⟨mld *example*⟩+≡
```
static void segment(int s) {
  switch(s) {
    case LIT:  set_block(rdata); break;
    case DATA: set_block(data);  break;
    case BSS:  set_block(bss);   break;
    default:   assert(0);
  }
}
```

4.10.3 Using relocatable addresses

`mld` associates a relocatable address with each label and global symbol. The target-independent procedure `global` defines a global symbol in the current data segment. If the symbol does not have a relocatable address, `mld` calls `addr_new` to create and initialize its address. An error message is issued if the symbol is already defined.

⟨mld *example*⟩+≡
```
static void global(p) Symbol p; {
```

[4] The current location in any text block should already be aligned on a 4-byte word. This is just a sanity check.

```
    ⟨assign p a relocatable address⟩
    if (!block_defined(p->x.raddr->label->block))
        label_define(p->x.raddr->label, p->x.raddr->offset);
    else  error("Attempt to redefine %s\n", p->name);
}
```

⟨*assign* `p` *a relocatable address*⟩≡

```
if (p->x.raddr == (RAddr)0)
    p->x.raddr = addr_new(label_new(p->name), 0);
```

References to global symbols often appear in data segments, e.g., as the initial values of other global symbols. `defaddress` emits the relocatable address of a symbol into the current data segment by calling `emit_raddr`, the constructor defined in Section 4.4 that emits a relocatable address.

⟨`mld` *example*⟩+≡

```
static void defaddress(p) Symbol p; {
    ⟨assign p a relocatable address⟩
    emit_raddr(p->x.raddr);
}
```

4.10.4 Relocation

When an encoding procedure emits an instruction that refers to an unknown relocatable address, it creates a closure to save the information necessary for relocating the instruction. `mld` supplies the procedure `mc_alloc_closure`, which creates a closure and saves it in a table associated with the current relocatable block in the current segment. The toolkit library initializes the returned closure.

⟨`mld` *example*⟩+≡

```
RClosure mc_alloc_closure(size_in_bytes, dest_block, dest_lc)
    unsigned size_in_bytes; RBlock dest_block; unsigned dest_lc; {
    RClosure cl;

    cl = (RClosure)mc_alloc(size_in_bytes, RClosure_pool);
    assert(dest_block == segments[cur_seg].relocblocks[segments[cur_rb]);
    addtotable(RClosure *, &(segments[cur_seg].closures[cur_rb]), cl);
    return cl;
}
```

The choice of when to perform relocation is application dependent. `mld` performs relocation after it has assigned an absolute address to all of the data segments and procedures and just before emitting an executable program. The toolkit provides most of the support for performing relocation in `apply_closure`. `mld`'s `dispatch` procedure applies `apply_closure` to every closure it has saved. If any relocatable address is unknown, `apply_closure` calls `error` to issue an error message. The implementations of `save_closure` and `dispatch` are target independent.

⟨mld *example*⟩+≡
```
void dispatch() {
    int seg, rb, i;
    RClosure cl;

    for (seg = 0; seg < NSEGS; seg++)
        for (rb = 0; segments[seg].relocblocks[rb] != 0; rb++)
            gentable(segments[seg].closures[rb], i, 0) {
                cl = tableentry(RClosure *, segments[seg].closures[rb], i);
                apply_closure(cl, cl_emitm, &error);
            }
}
```

4.10.5 Example makefile

The following example makefile shows how `tk.o`, an object file containing all of the encoding procedures needed by `mld`, is created. `mips.tk` is created by `tools` with `mips.spec`, the specification provided with the toolkit, and `mld-mips.spec`, a specification file specific to `mld`, are provided as inputs. The implementation of the encoding procedures are extracted from `mips.tk` into `tk.c` using `notangle`; the interface prototypes are extracted into `tk.h`. The library interface file, `mclib.h`, is included by `tk.c`.

```
mips.tk: mips.spec mld-mips.spec
    tools -byteorder fast -lc-cons-names -encoder mips.tk \
        mips.spec mld-mips.spec

tk.c: mips.tk
    notangle -t8 -L mips.tk | cpif tk.c

tk.h: mips.tk
    notangle -t8 -L -Rheader mips.tk | cpif tk.h

tk.o: tk.c tk.h mclib.h mldtk.h
    cc -c -I./ tk.c
```

4.11 Emitting assembly language

The details of getting the generator to create encoding procedures that emit assembly language are covered in Section 4.1. A toolkit application must provide a `printf`-like procedure plus a procedure for printing relocatable addresses:

⟨*library procedures*⟩+≡
```
extern void (*asmprintf)(void *closure, const char *fmt, ...);
extern void *asmprintfd; /* closure */
extern void (*asmprintreloc)(RAddr reloc); /* calls asmprintf */
```

The toolkit initializes these variables to print something sensible on standard output. Applications can change behavior by overwriting them.

Chapter 5

The generator and translator

Although we describe the generator and translator as separate entities, they actually are implemented by a single program, named `tools`. `tools` parses one or more files containing specifications and reports any syntactic or semantic errors. It is not unusual to decompose a machine specification into multiple files. `mld`, for example, generates encoding procedures using a "core" specification, a specification of "synthetic" instructions, and a specification of constructors used only by `mld`. Command-line options are used to ask `tools` to run the generator or translator or to perform other functions. Given no options, `tools` simply looks for syntactic and simple semantic errors in a specification.

The options recognized by `tools` are grouped into five categories:

E	Encoding
D	Decoding
d	Debugging decoding
V	Verbosity
A	Analysis

Table 5.1 shows all the options in alphabetical order.

5.1 E: Encoding options

`-asm-encoder` *file*

Like `-encoder`, but the generated encoding procedures use `asmprintf`, `asmprintfd`, and `asmprintreloc` to emit assembly language instead of binary. Puts the implementation into *file*`.c` and puts suitable declarations into *file*`.h`.

Option	Category	Usage
`-alt`	d	Show alternates at decision-tree leaves
`-ascii-dag` *file*	d	write an ascii representation of decision DAGs used in matching statements
`-ascii-tree` *file*	d	write an ascii representation of decision trees used to build matching DAGs
`-asm-encoder` *file*	E	generate assembly-emitting encoding procedures on *file*
`-asm-grammar` *file*	?	generate assembly-language grammar on *file*
`-byteorder` *string*	E	choose emission procedure
`-c`	DE	generate C code
`-checker` *file*	A	generate a checker on *file*
`-count`	E	Count invocations of encoding procedures
`-data`	E	Encoding procedures return data
`-debug-bal` *file*	d	write extra debugging information about the balancer
`-debug-checker` *file*	d	write information about generating checker inputs
`-debug-heur` *file*	d	write information about heuristic scores for tree-building
`-debug-match` *file*	d	write other information about building matching statements
`-debug-solver`	V	turn on solver debugging (to `stderr`)
`-debug-split` *file*	d	write debugging info when splitting tree nodes
`-decoder` *file*	D	from info in `-matcher` file, generate decoder on *file*
`-dis` *file*	D	generate disassembly code on *file*
`-dot` *file*	d	write `dot` commands showing matching statements
`-emit-bits` *n*	E	unit of emission is *n* bits
`-encoder` *file*	E	generate encoding procedures on *file*`.[ch]`
`-expand-spec` *file*	A	write expanded specification on *file*
`-fieldnames` *file*	A	write arrays of field names on *file*
`-foldemit`	E	fold constants in calls to emit procedure
`-icon`	D	generate Icon code
`-icon-dag` *file*	d	write an Icon representation of matching DAGs
`-icon-tree` *file*	d	write an Icon representation of decision trees used to build matching DAGs
`-impossible`	V	make errors force stop with stack trace
`-indirect` *name*[`:`*type*]	E	generated encoding procs called indirect through *name*
`-instype` *name*	E	Use *name* to refer to instructions
`-late-const`	E	use constants in closure functions (default)
`-late-none`	E	include no constants in closure functions
`-late-zero`	E	use zeroes in closure functions
`-lc-cons-names`	E	use all lower case for constructor names
`-lc-pat-names`	D	force pattern names to lower case
`-ledger`	d	Use ledger paper with `-dot`
`-matcher` *file*	D	transform pattern-matching statements in *file*
`-max-decimal` *n*	DE	values larger than *n* will be printed in hex
`-m3`	D	generate Modula-3 code
`-no-asm-ws`	E	remove whitespace from assembly syntax
`-no-pp`	E	don't prettyprint output; use $ escapes
`-no-reloc`	DE	make all relocatable addresses integers
`-nowarn`	V	suppress warnings (not recommended)
`-old-closures`	?	use the old closure technique (not recommended)
`-postfix`	ED	use postfix assembly-language syntax
`-symdis` *file*	E	put symbolic disassmbler on *file*.m and *file*.h
`-token-closures`	?	use one closure per token (recommended)
`-tryall`	d	try all heuristics on every field in tree
`-verbose`	V	print extra warnings and information

Table 5.1: Alphabetical list of all the options

`-byteorder` *string*

This misleadingly named option actually determines the name of the function that the encoding procedures use to emit tokens (`emit`*string*). Since the toolkit library provides `emitb`, `emitl`, and `emitm` procedures, which emit tokens using big-endian, little-endian, or machine-dependent byte orders, the values `b`, `l`, and `m` are good choices. My favorite trick, however, is to use `-byteorder X`, then use the C compiler to pick a byte order later on, for example, by compiling with `cc -DemitX=emitfast`. This trick avoids running the generator multiple times, which is good, because running the generator can be expensive.

`-c`

Tells the generator to generate C code (the only option), or tells the translator that the code emitted for the `-decoder` file should be C code.

`-count`

Tells the generator to include code in encoding procedures that counts the invocations of such procedures. This is useful for determining the static frequency of an instruction in a program, e.g., how many loads are generated. The C procedure

```
void reset_cons_counters(void);
```

resets the counters; the procedure

```
typedef void (*Printer)(void *closure, char *fmt, ...);
void dump_cons_counters(Printer p, void *closure);
```

dumps the values of all the counters. The user must provide a function of type `Printer`, a variadic function which accepts a closure, a `printf`-style format, and additional arguments. The closure encapsulates any information needed by the user's printing function. For example, the standard C function `fprintf` may be used as a `Printer`, in which case a file pointer acts as a closure.

`-data`

Have encoding procedures return data representing instructions at an abstract level. This is not the default; the default is for them to emit binary as a side effect.

`-emit-bits` *n*

The toolkit's generator works with bits, but it is more convenient for the toolkit's library to work with larger units. In particular, the second argument to the `emit` functions tells the library how wide the emitted token is, in units of n bits. The default is 8, which means that the `emit` functions work with bytes, since these are how the library distributed with the toolkit works. The width of every emitted token must be a multiple of n or the generator will complain.

`-encoder` *file*

Invokes the generator, which writes encoding procedures into *file*`.c` and puts suitable declarations into *file*`.h`.

`-foldemit`

Normally, the toolkit refrains from folding constants when generating the code that emits a token. This behavior makes it easy to examine the generated code to double-check that it really looks right. The `-foldemit` option tells the toolkit to fold constants aggressively. Its only real use is to make the generated code smaller, because modern compilers fold constants themselves, don't they?

`-indirect` *name*[`:`*type*]

Normally, the toolkit generates encoding procedures that are externally visible, that is, they occupy top-level name space. The `-indirect` option tells the toolkit to make the procedures local to a compilation unit and to provide indirect access to them through a structure holding procedure pointers. *name* gives the name of the structure, *type* its tag. (If *type* is omitted, we reuse *name*.) Useful for avoiding name clashes or for having multiple sets of encoding procedures in a single application.

`-instype` *name*

When return instructions as abstract data (instead of emitting bits) use *name* to refer to instructions.

`-late-const`

A heuristic that attempts to make closures smaller at the risk of creating more closure functions, by putting literal constants in the function bodies instead of the closures. This is the default.

`-late-none`

Do not apply either the `-late-const` or `-late-zero` heuristics. Makes closures larger but may create fewer closure functions. You probably don't want this.

`-late-zero`

A heuristic that attempts to make closures smaller at the risk of creating more closure functions, by putting literal zeroes in the function bodies instead of the closures.

`-lc-cons-names`

Ensures that only lower-case letters are used in the names of generated encoding procedures.

`-max-decimal` n

When generating code, the toolkit will use hexadecimal notation for integer literals larger than n. For smaller literals, it uses decimal notation. n must be nonnegative, so forcibly all negative integer literals are written in decimal notation. Default value is $2^{31} - 1$.

`-no-asm-ws`

Removes all whitespace from between operands in assembly-language syntax. This might be a cheap way to get compatibility with other tools.

`-no-pp`

This turns of the toolkit's internal prettyprinter; instead, it writes output that has a bunch of `$` escapes indicating how it should be printed. This is useful only if you intend to run the output through another prettyprinter.

`-no-reloc`

Forces all relocatable operands to be integers. Useful when generating encoding procedures to be called from a disassembler.

`-postfix`

For disassembly or for assembly-language encoding, put opcode *after* operands.

`-symdis` *file*

To *file*`.h`, write type declarations of a symbolic representation of instructions. To *file*`.c`, write a disassembler that returns that representation.

5.2 D: Decoding options

`-c`

Tells the generator to generate C code (the only option), or tells the translator that the code emitted for the `-decoder` file should be C code.

`-decoder` *file*

Invoke the toolkit translator, which translates embedding matching statements from the `-matcher` file and writes the results on the `-decoder` file. To use the translator, both `-matcher` and `-decoder` must be given exactly once.

`-dis` *file*

Generate code for disassembly on *file*. In flux right now.

`-icon`

Tells the translator to attempt to generate encoding procedures written in Icon. Not really supported.

`-lc-pat-names`

Pattern names are accessible using the `[`*name*`]` syntax in matching statements; this option forces all such names to use lower-case letters only.

`-matcher` *file*

Invoke the toolkit translator, which translates embedding matching statements from the `-matcher` file and writes the results on the `-decoder` file. To use the translator, both `-matcher` and `-decoder` must be given exactly once.

`-max-decimal` *n*

When generating code, the toolkit will use hexadecimal notation for integer literals larger than n. For smaller literals, it uses decimal notation. n must be nonnegative, so forcibly all negative integer literals are written in decimal notation. Default value is $2^{31} - 1$.

`-m3`

Tells the translator that the code emitted for the `-decoder` file should be Modula-3 code. (The toolkit generator and library do not yet support Modula-3.)

`-no-reloc`

Forces all relocatable operands to be integers. Useful when generating encoding procedures to be called from a disassembler.

`-postfix`

For disassembly or for assembly-language encoding, put opcode *after* operands.

5.3 d: Options for debugging decoders

These options provide different kinds of diagnostic information that shows how the toolkit is building a decoder.

`-alt`

Prints something to do with alternatives at the leaves of decision trees? I'm no longer sure...

`-ascii-dag` *file*

Writes on *file* an ASCII representation of the decision trees (actually DAGs) used to implement matching statements. Useful for debugging, especially performance debugging. *Probably broken. In any case, it should be rewritten to use the new prettyprinter.*

`-ascii-tree` *file*

Writes on *file* an ASCII representation of the decision trees before they are converted to DAGs. This option is really here only in case something goes badly wrong with the dagging process (or in case you're curious). *This, too, is probably broken.*

`-debug-bal` *file*

This option causes more information about the operation of the balancer to be written to *file*, as an aid to debugging unsolvable sets of equations.

`-debug-checker` *file*

To produce a checker program, the toolkit attempts to generate test inputs that will satisfy every branch of every constructor. This option writes information about choosing test values onto *file*.

`-debug-heur` *file*

When building the decision tree, the toolkit uses about a half dozen heuristics to choose the field to test next. Each heuristic assigns a score to each field. This option writes information about heuristic scoring onto *file*.

`-debug-match` *file*

During tree building, writes information about building matching statements onto *file*.

`-debug-split` *file*

During tree building, writes information about splitting internal nodes onto *file*.

`-dot` *file*

Creates output for the `dot` graph-drawing program, which can be used to create a graphical representation of the decision tree (actually a DAG) used to implement matching statements. The output is written to *file*. *This option probably doesn't work. It certainly doesn't understand multiple branches with identical patterns but different conditions.*

`-icon-dag` *file*

Writes on *file* an Icon representation of the DAGs used to implement matching statements. This code can be used to create a CGI program that lets you browse trees with Netscape.

`-icon-tree` *file*

Writes on *file* an Icon representation of the decision trees before they are converted to DAGs. Most people will use `-icon-dag` instead.

`-ledger`

Insert a PostScript incantation that might make the `-dot` option use ledger paper instead of regular paper. *Almost certainly broken.*

`-tryall`

Show what split would be given by each heuristic. Useful only with the option `-debug-heur`.

5.4 V: Verbosity options

These options control the verbosity of the messages printed by the toolkit.

`-debug-solver`

Makes the equation solver (and a few other parts of the toolkit) print lots of debugging information on standard error.

`-impossible`

Arranges for all error messages to be treated as "impossible," which causes "this can't happen" to be printed and a stack trace to be generated. It is intended primarily for debugging the toolkit.

`-nowarn`

Suppresses warnings. It's not a good idea to use this option routinely. If a warning annoys you, let us know—maybe we can make it appear only when `-verbose`.

`-verbose`

Make the toolkit print extra warnings and informational messages. We've tried to include enough stuff so that you can watch the toolkit's progress through a big specification, even on a slow computer.

5.5 A: Analysis options

These options support neither encoding nor decoding; instead, they offer various kinds of analysis of what's in the toolkit specification. In fact, it's even possible to run `tools` without any options at all, just to check the syntax of a specification.

`-checker` *file*

Writes the source for a checker in *file*. For each instruction in a specification, the checker program emits assembly code and binary code in ASCII into an assembly file. The resulting file is then assembled and disassembled. The disassembled output should contain pairs of matching instructions.

`-expand-spec` *file*

Rewrites the specification onto *file*, but with all patterns in normal form. Can be useful for seeing the meanings of complex opcodes. *This option predates the introduction of constructors, which it does not treat, and it may break horribly. Pester us if you want it brought up to date, so that it emits a complete specification in some canonical form.*

`-fieldnames` *file*

> For each field in the specification, write to *file*`.c` an array of string literals representing the names of values of that field. Write declarations to *file*`.h`. Such arrays may be useful in writing disassemblers. (Will use extensions `.i3` and `.m3` if `-m3` is given.)

5.6 Example execution of `tools`

The `tools` program is a prototype and has been used by few people other than its authors. We believe we have excised all uninteresting diagnostic messages; let us know if you think otherwise. The line numbers used in error messages reflect the state of the parser when the error is detected, not where the error actually occurred. In particular, many semantic errors are not detected until the whole specification is parsed, in which case the line number is worse than meaningless; it is misleading. We apologize for this behavior, which we hope to correct in a future release.

5.6.1 Running the generator

The following command uses the specification in `mips.spec`. Because the `-m3` option is not used, the toolkit generates C code; it writes an encoder to the file `mipstk.c`, and it writes a header file to `mipstk.h`. Commands typed by the user are shown in *`typewriter italic`*; the toolkit's responses are shown in `typewriter`. Wide lines have been broken by hand.

```
% tools -byteorder X -encoder mipstk mips.spec
Parsing...
Done parsing.
```

5.6.2 Running the translator

To generate a decoder, we use two specifications: `mips.spec`, as in the generator command, and `fetch.spec`, which specifies how to fetch words from an instruction stream. `dis.m` includes the matching statement for the decoding application; the decoding code is emitted in C to `dis.c`.

```
% tools -c -decoder dis.c -matcher dis.m mips.spec fetch.spec
Parsing...
Done parsing.
```

Chapter 6

Validating a specification

It doesn't do any good to generate tricky bit-fiddling code from a specification unless the specification itself is accurate. The toolkit warns users of certain suspicious specifications, like forgetting to use an operand in a constructor definition, and it balks at outright erroneous specificiations, operations, like conjoining overlapping fields or by assigning a field two different values, but the toolkit does not and *cannot* guarantee that a specification accurately describes the target machine. Errors that cannot be caught be the toolkit include getting opcodes out of order in a table or getting the order of operands wrong in a constructor definition. To catch these kinds of errors, the toolkit provides a facility to *validate* a specification against an existing assembler or disassembler. We suggest that you validate new specifications before using toolkit-generated code in an application, because debugging a specification in the context of a large application is tedious.

To identify possible errors in a specification, the toolkit can generate a specification "checker." Figure 6.1 depicts the mappings performed by a checker. For each untyped constructor, i.e., instruction, in a specification, a checker emits its representation in assembly ($A.instructions$) and its representation in binary ($A.data$). A constructor's assembly representation is derived from its left-hand side and from assembly-language syntax specifications as described in Section 2.10. For convenience, the checker emits each constructor's "binary" representation in assembly language, using directives like `.byte` and `.word`. This trick enables the checker to write both representations in one assembly file. For example, a checker for the Pentium emits the code in the left column of Table 6.1 to test the instruction `addb`. The "assembly" version of the instruction uses the usual syntax, and the "binary version," which occupies two 8-bit tokens, is specified with two `.byte` directives.

The assembly code is assembled by a vendor's assembler into object code, which produces two binary representations of each constructor ($B'.instructions$ and $B''.instructions$, center column in Table 6.1). This binary representation is good enough to detect errors—the checker could compare binary representations in an object file—but just knowing that two binary representations differ

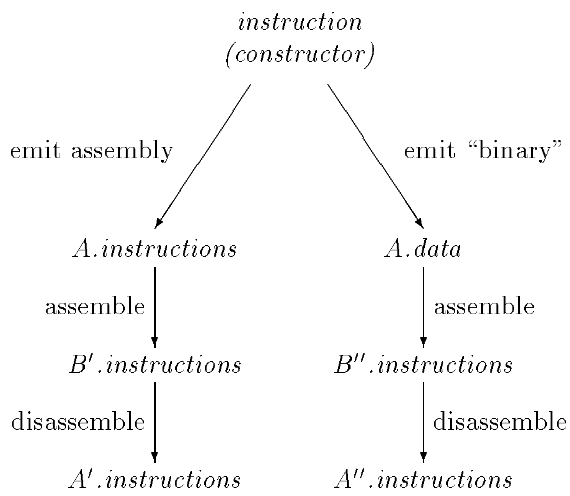

Figure 6.1: Mappings performed by toolkit specification checker.

Emitted	Assembled to	Dissassembled to
`addb $127,%al`	0x047f	`addb $127,%al`
`.byte 0x4`	0x047f	`addb $127,%al`
`.byte 0x7f`		

Table 6.1: Output of toolkit checker.

doesn't provide any useful hints about the likely cause of the error. To help users understand errors, the checker invokes a vendor's disassembler to disassemble the object code into pairs of assembly instructions (A'.*instructions* and A''.*instructions*, right column in Table 6.1). If the specification is consistent with the assembler, the output from the disassembler should contain pairs of matching instructions; a matched pair indicates that the assembler's binary encoding of an instruction matches the toolkit's encoding. The shell script `chkrfilter` compares pairs of instructions and flags any mismatched pairs. Because mismatched pairs are displayed in assembly language, the user can easily identify the inconsistency in the specification.

The operand values used to test constructors are generated automatically by the toolkit. For each branch of each constructor, the toolkit attempts to generate one tuple of operand values that satisfies that branch. The checker program applies the constructor to each generated tuple of operand values, with the goal of exercising each branch once. In general, it is not possible to guarantee that every branch will be exercised, for example when a branch's equations contain mutually unsatisfiable constraints, such as `{rd = 1, rd = 2}`. Even if a tuple of operand values satisfies the i-th branch, B_i, in the ordered set of a constructor's branches, it is possible B_i will not be exercised, when the tuple also satisfies some branch that occurs before B_i, i.e., $B_1 ... B_{i-1}$. Although the toolkit cannot guarantee that every branch will be exercised, it does flag those branches for which it cannot compute a operand tuple guaranteed to exercise that branch.

6.1 Generating a checker program

We have validated the MIPS and SPARC specifications and the 32-bit subset of the Pentium specification that are distributed with the toolkit. Validation found several errors in each target's specification. To generate and run a checker for one of the supported targets, execute `make` *target* in the `base-checker` directory. The `makefile` contains the rules for building and running a checker. The rest of this section describes how a checker is built and run.

The toolkit-generated program "checks" a specification by applying pairs of constructors for each instruction. The first constructor emits the instruction's binary code. For convenience, a constructor's binary representation is emitted using assembly directives (e.g., `.byte, .word, .long`), which allows the checker to emit both representations in one assembly file. The second constructor in a pair emits the the assembly-language representation of the instruction. When executed, the checker program emits its output into an assembly file, which is then assembled by a vendor's assembler. This technique assumes that the vendors's assembler permits both instructions and data in the text segment; the SGI-Irix assembler is the only one that we have found that prohibits this mix.

The resulting object file is disassembled using a native disassembler or debugger. We use `gdb -batch` to disassemble the object file. `gdb` reads the files `gdb-{asm,bin}output`, which disassemble the instructions generated by the checker.

If the specification's assembly-to-binary mappings are correct, the disassembled output should contain pairs of matching instructions; a matched pair indicates that the assembler's binary encoding of an instruction matches the toolkit's encoding. The script `chkrfilter` compares pairs of instructions and flags any mismatched pairs. Because the checker emits mismatched pairs in assembly, the user can easily identify the source of the discrepancy in the original specification.

The sample `makefile` shows all the steps necessary to build and run a checker. A checker requires an assembly emitter (*target*`-asm.c`), a binary emitter (*target*`-bin.c`), target-specific code to emit bytes in assembly pseudo-ops (*target*`-bits.c`), the checker program (*target*`-check.c`), and the toolkit library. The checker program accesses the binary and assembly emitters indirectly via the interfaces *target*`_asm` and *target*`_bin`.

To check instructions that take relocatable operands, we use a trick to guarantee that the address assigned by the assembler matches that used by the binary emitter. The address in the file `chkraddr` is the entry point for `main`; we get that value before building the checker program, then use it to assign known addresses to relocatable operands. The assembly directives for emitting bytes on the supported targets are in `emitbits.nw`.

⟨*sample rules to build checker*⟩+≡

```
chkraddr:
        echo "main(){}" >_boot.c
        $(CC) -g _boot.c -o _boot
        nm _boot | egrep "T _main|T main" | \
                   awk '{print "0x"$$1}' | cpif $@
        rm -f _boot.c _boot
```

A specification is checked in several steps. All of these steps appear in the `base-checker`'s makefile. First, the checker program emits assembly code in the file *_target.s*. The assembly file is assembled into the binary file *_target.out*. Then the `chkrfilter` shell script disassembles the binary file and compares the two representations. Any differences are printed to standard output.

6.2 Building a checker for a supported target

To try a checker program, execute `make` *target*, e.g., `make sparc`, using the version 0.4 `makefile` in the directory `base-checker`. An executable checker will be created in *target*`-checker`. That program is executed and its output is checked for inconsistencies. You *must* have `gdb` to disassemble the output. If you don't have `gdb`, any disassembler will probably do, but you will have to edit `chkrfilter` and make the appropriate substitutions for `gdb`.

The MIPS and SPARC specs can be checked in their entirety. The Pentium spec takes too long to process so at once, so we check it in parts: several sets of instructions in alphabetical order, then the arithmetic, and then the floating-point instructions.

Appendix A

Vaporware

This appendix describes procedures that we would like to have in the library but that we haven't had time to implement. These procedures are geared to supporting *toolkit object code*, viz., a disk-based representation of the data structures used in encoding applications. We intend to use *tables* from Hanson (1996) so that labels and such can be referred to by name, and we need to implement *pickles* to be able to read and write relocatable blocks, relocation closures, and so on.

A.1 Tables

Applications will need tables for binding names of objects to their values. The library provides two predefined, global tables:

`labeltab`	Binds the names of global labels to the labels themselves
`sectiontab`	Binds the names of sections to lists of relocatable blocks

An assembler may use `labeltab` to store labels defined in assembly text and `sectiontab` to store relocatable blocks that represent data sections or procedure text. The application create more tables.

The application must tell the library what section names it understands. `newsection` creates a new section if it has not already been created. An assembler, for example, might call `newsection` when assembling a pseudo-op such as ".data" for the first time.

⟨*library*⟩≡
```
procedure newsection(char *name)
    /sectiontab[name] := [] | fail(multiply defined sections)
```

Each entry of `sectiontab` is a list. Relocatable blocks are added to a section by simply concatenating the block on the appropriate list. A C implementation of the library will provide the C equivalents of the Icon code `l := [], put(l,x), every x := !l do ...`.

A.2 Pickling and Unpickling

The application can externalize and internalize all of the data structures provided by the library using the following "pickling" and "unpickling" interface.

The following EBNF grammar specifies the layout of pickled objects on disk.[1] The smallest unit of reading and writing is a word. Within the grammar, four-letter strings, integer literals, and symbols in all caps are terminal symbols standing for single words. Symbols in mixed case are nonterminals.

The word represented by a four-character string is computed by putting the first character of the string in the most significant byte, and so on. For example, the string `"tabl"`, used to tag tables, forms the type tag

```
't' << 24 + 'a' << 16 + 'b' << 8 + 'l'.
```

This type tag *cannot* always be formed by casting the string `"tabl"` to an integer using the native byte order of the machine.

The first word in a pickle is always `"picl"`. By checking this word, a program reading the pickle can tell whether the words in the pickle were written in the native or swapped byte order. A pickle always represents a single object. To provide a modicum of error checking, the object is always tagged. Most applications pickle their own data structures, which may point to more than one object of the kind defined in this library.

⟨*pickle grammar*⟩≡
```
Pickle  => "picl" Tagged
```

Every object within a pickle is identified by a unique signed integer or *UID*; UIDs make it possible to represent cycles and pointer sharing. The UID 0 always represents the `NULL` or `NIL` pointer.

Under some circumstances, a UID may be followed by a *type tag*, which identifies the type of the object. For example, every element of a heterogeneous list must carry a type tag. This library reserves the type tags in the range 0–32767 for use by applications.

The following productions show the type tags reserved for use with the toolkit types. The library supports only homogeneous and lists, and homogeneous tables keyed by strings.

⟨*pickle grammar*⟩+≡
```
Any    => [ Table | List | RLabel | RAddr | RBlock
          | RClosure | INTEGER | UserDef
          ]
Tagged => [ "tabl" Table | "list" List | "rlbl" Rlabel
          | "radr" RAddr | "rblk" RBlock | "rclo" RClosure
          | "chrs" String | "int\0" INTEGER | TAG UserDef
          ]
```

[1] In EBNF, | denotes alternation, `[...]` optional fields, and `{...}` Kleene closure.

A pickled table includes a `TAG` denoting the type of the table's elements, a `Count` denoting the total number of table elements, and a sequence of key-value pairs. The values are untagged because the single `TAG` at the beginning describes all of the values.

⟨*pickle grammar*⟩+≡
```
Table => UID [TAG Count {String Any}]
```

A `List` is just like a table but does not include any keys.

⟨*pickle grammar*⟩+≡
```
List     => UID [TAG Count {Any}]
```

Of the four basic data types provided by the library, `RAddr`, `RLabel`, `RBlock`, and `RClosure`, the first two are completely straightforward:

⟨*pickle grammar*⟩+≡
```
RAddr   => UID [OFFSET RLabel]
RLabel  => UID [OFFSET RBlock]
```

An `RBlock` contains two optional fields `ADDRESS` and `REG OFFSET`. The value of the `Flags` determines the presence of these fields. If `Flags & 1`, `ADDRESS` is the value of `rb.address`; otherwise, `ADDRESS` is omitted and `rb.address` is null. *and similarly for other fields...* `String` denotes `rb.contents`.

⟨*pickle grammar*⟩+≡
```
RBlock  => UID [LOW RLabel FLAGS [ADDRESS] [REG OFFSET] String]
```

An `RClosure` includes a `Function`, which represents the closure's `apply` method. `COUNT` denotes the number of relocatable addresses in the closure. Since a closure is created only when a relocatable address is unknown, there must be at least one `RAddr` in the closure.

⟨*pickle grammar*⟩+≡
```
RClosure => UID [OFFSET Function RBlock COUNT {RAddr}]
```

A function is represented by bytecodes. So far, the only defined "bytecode" is actually a PostScript-like "string code" described in Ramsey (1996). The toolkit generates these codes when it generates closure functions. This code is indicated by the tag `"tkps"`.

⟨*pickle grammar*⟩+≡
```
Function => "tkps" String
```

Strings are an exception to the rule that all reading and writing is word-oriented. The disk representation of a string is a word indicating the number of bytes in the string, followed by the bytes themselves. The bytes are followed by zero or more zero bytes, used to pad the string so its length is a multiple of the word length. This representation is *not* equivalent to the C representation; for example, strings may contain embedded null characters. The C version of the library provides a procedure that pickles and unpickles strings in this format.

⟨*pickle grammar*⟩+≡
```
String => UID [COUNT bytes padding]
```

A.3 C language support for pickles

To pickle and unpickle a type, the library needs a tag, a pickling procedure, and an unpickling procedure:

⟨*C library*⟩≡
```
typedef struct pklstream *PklStream;
unsigned getword(PklStream s);                /* probably a macro */
procedure putword(PklStream s, unsigned n);   /* probably a macro */
procedure read_bytes(char *buf, int size, PklStream s);

typedef struct pkltag {
    unsigned tagword;   /* TAG */
    void (*write)(PklStream s, void *data, ...);
    void *(*read)(PklStream s);
} *PklTag;
```

All calls to write procedures must be bracketed between `pickleBegin` and `pickleEnd`.

A `PklTag` is provided for each library data structure. The write procedures for homogeneous tables and lists take an extra argument, which is a tag describing the element type.[2] Pickling procedures that are part of the application may use this extra argument to pickle tables and lists of the application's own data structures. This library provides no support for pickling and unpickling heterogeneous tables and lists.

⟨*C library*⟩+≡
```
PklTag pickleTab, pickleList, pickleRAddr, pickleRBlock,
       pickleRClosure, pickleRLabel;
```

As an example, here is the code used to pickle a table `d` in which the elements are labels:

⟨*example use*⟩≡
```
pickleTab->write(s, d, pickleRLabel);
```

`pickleRBlockList` is provided as a tag that simplifies pickling of `sectiontab`. To pickle `sectiontab`, the application would call

`pickleTab(sectiontab, pickleRBlockList)`.

⟨*library*⟩+≡
```
PklTag pickleRBlockList;
    pickleRBlockList->write(s, l) == pickleList->write(s, l, pickleRBlock)
```

Applications use the tags to write and read pickles. `pickleWrite(fd, p, tag)` pickles the object pointed to by `p`, pointers within the object are followed and their referents pickled, and so on transitively. Pointer cycles and sharing are correctly represented. *flag for tagging every single object?*

[2] This admittedly ugly scheme is a poor man's substitute for real closures.

⟨*C library*⟩+≡

```
procedure pickleWrite(int fd, void *p, PklTag tag);
    write disk representation of p to descriptor fd.
```

The library uses `int fd` instead of `FILE *fp` for input and output because some applications (like `lcc` and `mld`) don't tolerate `FILE *`.
`pickleRead` reads a pickled object from disk. *Support for pickles in which all objects are tagged?*

⟨*library*⟩+≡

```
procedure *pickleRead(int fd, PklTag type_expected, Arena *p,
                 (char *)(*pickleGetmem)(int size), int initialSize);
    return pointer to object in pickle
```

Allocation of pickles is arena-based, and all objects in a pickle are allocated in the same arena. If `*p` is nonzero, it is used as the arena; otherwise a new arena is created and *p is set to it. If `getmem` is nonzero, it is used to add space to the arena. If `pickleSizeHint` is nonzero, it is used as the initial size of the arena when an arena must be created.

References

Fraser, Christopher W. and David R. Hanson. 1991 (October).
A retargetable compiler for ANSI C.
SIGPLAN Notices, 26(10):29–43.

Fraser, Christopher W., Robert R. Henry, and Todd A. Proebsting. 1992 (April).
BURG—fast optimal instruction selection and tree parsing.
SIGPLAN Notices, 27(4):68–76.

Hanson, David R. 1996.
C Interfaces and Implementations.
Benjamin/Cummings.

Joy, William. 1994.
An Introduction to the C shell, Chapter 4.
Sebastopol, CA: O'Reilly.

Milner, Robin, Mads Tofte, and Robert W. Harper. 1990.
The Definition of Standard ML.
Cambridge, Massachusetts: MIT Press.

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

Ramsey, Norman and Mary F. Fernández. 1994 (October).
New Jersey Machine-Code Toolkit architecture specifications.
Technical Report TR-470-94, Department of Computer Science, Princeton University.
Revised December, 1996.

———. 1995 (January).
The New Jersey Machine-Code Toolkit.
In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

———. 1997 (May).
Specifying representations of machine instructions.
ACM Transactions on Programming Languages and Systems, 19(3):492–524.

Index