# The New Jersey Machine-Code Toolkit

Norman Ramsey
*Bell Communications Research*
Mary F. Fernandez
*Department of Computer Science, Princeton University*

## Abstract

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

The toolkit's specification language uses four concepts: *fields* and *tokens* describe parts of instructions, *patterns* describe binary encodings of instructions or groups of instructions, and *constructors* map between the assembly-language and binary levels. These concepts are suitable for describing both CISC and RISC machines; we have written specifications for the MIPS R3000, SPARC, and Intel 486 instruction sets.

We have used the toolkit to help write two applications: a debugger and a linker. The toolkit generates efficient code; for example, the linker emits binary up to 15% faster than it emits assembly language, making it 1.7–2 times faster to produce an `a.out` directly than by using the assembler.

## 1 Introduction

The New Jersey Machine-Code Toolkit helps programmers write applications that process machine code—assemblers, disassemblers, code generators, tracers, profilers, and debuggers. The toolkit lets programmers encode and decode machine instructions symbolically. It transforms symbolic manipulations into bit manipulations, guided by a specification that defines mappings between symbolic and binary representations of instructions. We have written specifications for the MIPS R3000, SPARC, and Intel 486 instruction sets.

Traditional applications that process machine code include compilers, assemblers, linkers, and debuggers. Some applications avoid machine code by using assembly language; e.g., most Unix compilers emit assembly language, not object code. Often, however, it is not practical to use an assembler, as when generating code at run time or when adding instrumentation after code generation. Applications that work on object code are more widely useful than those that require assembly code or source code, because they can be used on any executable file. Our toolkit makes such applications easier to implement.

Currently, applications that can't use an assembler implement encoding and decoding by hand. Different *ad hoc* techniques are used for different architectures. The task is not intellectually demanding, but it is error-prone; bit-manipulating code usually harbors lingering bugs. Our toolkit automates encoding and decoding, providing a single, reliable technique that can be used on a variety of architectures.

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

The toolkit has three parts. The *translator* translates the matching statements in a C or Modula-3 program into ordinary code. The *generator* generates encoding and relocation procedures in C. The *library* implements both instruction streams and relocatable addresses, which refer to locations within the streams. The translator and generator need an instruction specification;

encoding procedures are generated from the specification, and matching statements can match the instructions or parts thereof defined in the specification. The library is machine-independent.

The toolkit's specification language is simple, and it is designed so that specifications resemble descriptions found in architecture manuals. It uses a single, bidirectional construct to describe both encoding and decoding, so their consistency is guaranteed. The toolkit checks specifications for unused constructs, underspecified instructions, and inconsistencies. An instruction set can be specified with modest effort; our MIPS, SPARC, and 486 specifications are 127, 193, and 460 lines.

We have two applications that use the toolkit. `mld` (Fernandez 1994), a retargetable, optimizing linker, uses the toolkit to encode instructions and emit executable files. `ldb` (Ramsey 1992; Ramsey and Hanson 1992), a retargetable debugger, uses the toolkit to decode instructions and to implement breakpoints.

The toolkit provides practical benefits, like reducing retargeting effort. For example, `ldb`'s disassembler for the MIPS requires less than 100 lines of code, and `mld` has replaced 450 lines of hand-written MIPS code with generated encoding and relocation procedures. By hiding shift and mask operations, by replacing case statements with matching statements, and by checking specifications for consistency, the toolkit reduces the possibility of error. The toolkit can speed up applications that would otherwise have to generate assembly language instead of binary code. For example, `mld` emits executable files 1.7 to 2 times faster by using encoding procedures than by writing assembly language and using native assemblers. Such speedups would otherwise require hand-written encoding and relocation procedures for each target architecture.

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

## 2 Using the toolkit

Figures 1 and 2 show how the toolkit is used in two applications. Code is shown in boxes, data in ovals. Code in doubled boxes is machine-dependent; a version exists for each target architecture. Code in single boxes is machine-

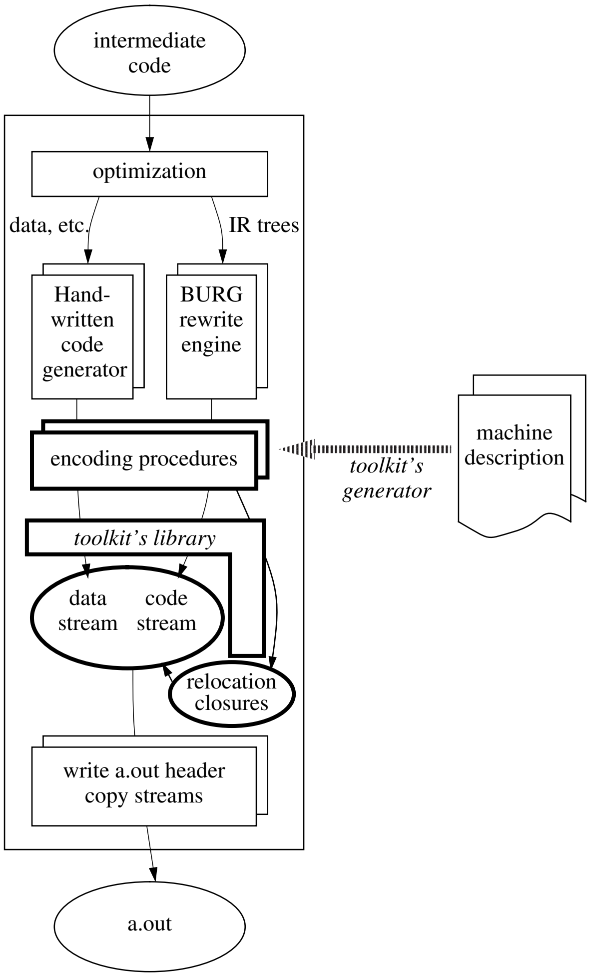


Figure 1: Structure of `mld`

independent. Code pointed to by thick, dashed arrows is generated by the toolkit. Boxes with heavy borders contain code that is part of the toolkit or generated by the toolkit. Ovals with heavy borders contain instruction streams that are written or read by toolkit-generated code. The names of the three parts of the toolkit are shown in italics.

### `mld`

`mld`, shown in Figure 1, is a retargetable, optimizing linker for the MIPS and SPARC. `mld` links a machine-independent intermediate code, optimizes it, generates instructions and data, and emits a machine-dependent executable file (`a.out`). Retargeting `mld` requires adapting a code generator and writing code to emit an `a.out` file.

`mld`'s code generators are based on those used in the `lcc` compiler (Fraser and Hanson 1991), which emit assembly code. Much of each `lcc` code generator is generated automatically from a BURG specification (Fraser, Henry, and Proebsting 1992), which rewrites 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 encoding procedures generated by the toolkit, e.g., the call

```
printf("fadds %%f1, %%f2, %%f3")
```

is replaced by the call `fadds(f1, f2, f3)`, where `fadds` is generated by the toolkit. The encoding procedures use the toolkit's library to emit code into an instruction stream in `mld`'s memory. The adapted code generators also call the library directly, e.g., to emit data.

`mld` uses instruction streams to model segments in an executable file. For example, a SPARC executable contains instructions in a "text" segment, initialized data in a "data" segment, and uninitialized data in a "bss" segment. `mld` uses one instruction stream for each segment. Instructions can be emitted into a stream even when they refer to labels whose positions are not yet known. The toolkit's generator creates code to resolve such references, so that `mld` needs only 20 lines of C code for relocation, and that code is machine-independent.

### `ldb`

`ldb`, shown in Figure 2, is a retargetable debugger for ANSI C. Most of its breakpoint implementation is machine-independent; the only machine-dependent part is the analysis of control flow (Ramsey 1994a). The analysis is written using a matching statement. The toolkit's translator can use any method of fetching instructions from streams; it works by instantiating templates that are supplied at translation time. For `ldb`, the templates fetch instructions from the program being debugged. The translator uses these templates and the SPARC instruction specification to transform the analysis's embedded matching statement into Modula-3 (Nelson 1991).

Matching statements make disassembly and flow analysis clear and concise. The analysis, for example, takes only a dozen lines of code; we present it in Section 9. Writing the analysis or disassembly by hand would require large nested case statements that enumerated all the SPARC opcodes.

## 3 Specification Concepts

Because machine instructions don't always fit in a machine word, the toolkit works with streams of instructions, not just the instructions themselves. An instruction stream is like a byte stream, except that the units may be "tokens" of any size. An instruction is a sequence of one or more tokens; for example, a 486 instruction might include several 8-bit prefixes, an 8-bit opcode, 8-bit format bytes, and a 16-bit immediate operand.

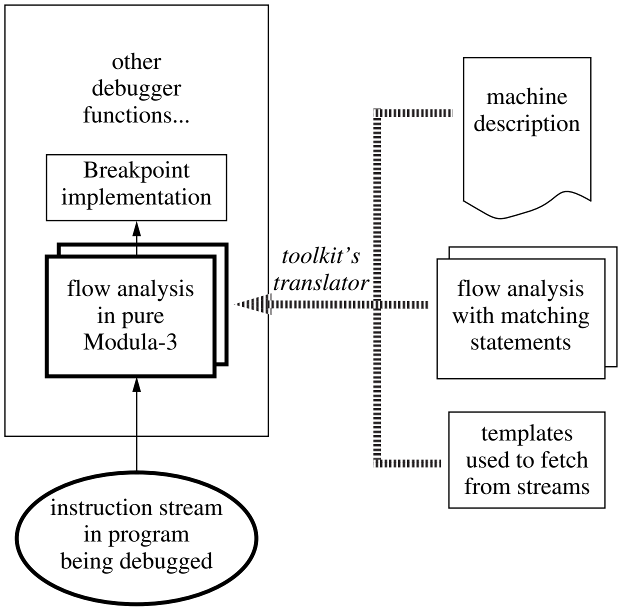


Figure 2: Use of decoding in `ldb`

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

*Constructors* connect the symbolic and binary representations of instructions. At a symbolic level, an instruction is an opcode (the constructor) applied to a list of operands. The result of the application is a sequence of tokens, which is described by a pattern. Application programmers use constructors to emit instructions, by calling procedures derived from constructor specifications, and to decode instructions, by using constructors in matching statements to match instructions and extract their operands.

## 4 Tokens and fields

`fields` declarations specify how to divide tokens into fields. One `fields` declaration is given for each *class* of tokens; only fields named in the declaration can be extracted from tokens of that class. The declaration binds field names to bit ranges and specifies the number of bits in tokens of its class. The toolkit generates the shifts and masks needed to get the value of a field in a token. Field values are always unsigned; a postfix exclamation point can be used to sign-extend them.

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

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

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

| op | rd | op3 | rs1 | opf | rs2 |
|---|---|---|---|---|---|
| 31 30 | 29 25 | 24 19 | 18 14 | 13 5 | 4 0 |

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

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

The first two indented lines define the fields used in the formats pictured above; the last two lines define fields used in SPARC formats that aren't pictured in this paper. Because all SPARC instructions are 32 bits wide, only one class of tokens is needed, the 32-bit `itoken` (for "instruction token"). When instructions vary in size, more classes may be needed. On the Intel 486, instructions are composed of 8-, 16- and 32-bit tokens, which must be given different classes because they are of different sizes. It can even be useful to put tokens of the same size in different classes. For example, the 486 uses a "ModR/M" byte to specify addressing modes and an "SIB" byte to identify index registers (Intel 1990, page 26-3):

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

The `fields` declarations for these bytes are:

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

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

# 5 Patterns

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

Patterns are composed from *constraints* on fields. A constraint fixes the range of values a field may have. The typical range has a single value, e.g., `op = 1`. Patterns may be composed by conjunction (`&`), concatenation (`;`), or disjunction (`|`).

The basic patterns and pattern operators are best understood by considering how they affect matching. The constraint "`lo <= f < hi`" on a field `f` 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 wild-card constraint "`some` *class*" matches any token of class *class*, for example, on the SPARC, "`some itoken`" matches any 32-bit token. A conjunction "`p & q`" matches if both `p` and `q` match.[1] 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.

## Patterns in Specifications

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

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

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

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

Defining opcodes individually would be tedious, and the result would be hard to compare with the architecture manual, which uses opcode tables. The `patterns` declaration can bind a list of names if a *generating expression* appears on the right. Generating expressions are modeled on expressions in the Icon programming language, which can produce more than one value (Griswold

[1] Conjunction is permitted if and only if the constraints that are conjoined refer to fields in tokens of the same class; this restriction enforces the rule against mixing fields from different classes of tokens. For example, on the 486, the pattern `mod = 0 & r_m = 5` is permitted, but the pattern `mod = 0 & index = 2` is not.

and Griswold 1990). A generating expression is a pattern in which some integers have been replaced by expressions in brackets like `{0 to 3}`, which denotes the sequence of integers $(0, 1, 2, 3)$. These expressions are activated in left-to-right LIFO order, resulting in a list of patterns, each of which is bound to the corresponding name on the left. For example, the following declaration describes the first opcode table in the SPARC manual (SPARC 1992, p 227):

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

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

Most manuals give tables in which not every opcode is used. Unused opcodes can be bound to the special name "_", which may be used only on the left side of a binding. For example, Table F-3 from the SPARC manual defines many of the arithmetic opcodes (SPARC 1992, p 228):

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

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

# 6 Constructors

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

To reduce the possibility of errors, a specification writer can associate a type with a constructor. Applying a constructor of type $T$ produces a value that is useful only as an operand to another constructor that expects an operand of type $T$. Applying an untyped constructor emits tokens into an instruction stream.

Because assembly language is the most familiar symbolic representation of instructions, we designed constructor specifications so their left-hand sides resemble assembly-language syntax: a constructor name and a list of operands. Operands may be separated by spaces, commas, brackets, or other punctuation. The punctuation is ignored; it is only syntactic sugar. The right-hand side of a constructor specification contains a pattern that describes the binary representation of the instruction specified. That pattern may contain free identifiers, which refer to the constructor's operands; such operands may be integers, or they may be patterns produced by constructors of a given type. For example, the following constructor describes the SPARC floating-point negate instruction:

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

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

```
patterns fnegs is fpop1 & opf = 0x5
```

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

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

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

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

```
constructors
  fnegs rs2, rd
```

Because no constructor type is given, fnegs is untyped. The generated encoding procedure, which has the C declaration

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

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

Not all operands are simple integers or fields. For example, the SPARC integer-arithmetic instructions take a second operand that may be a register or an immediate operand, depending on the value of the i field (SPARC 1992, p 84). Our SPARC specification defines the constructor type reg_or_imm for these operands, with register-mode and immediate-mode constructors:

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

where simm13! denotes the field simm13 interpreted as a signed integer. The identifier reg_or_imm can be used as an operand in the definitions of other constructors, for example

```
constructors
  add rs1, reg_or_imm, rd
```

The encoding procedures generated from these declarations are:

```
reg_or_imm_Instance rmode(unsigned rs2);
reg_or_imm_Instance imode(int simm13);
void add(unsigned rs1,
         reg_or_imm_Instance reg_or_imm,
         unsigned rd);
```

The rmode and imode procedures have no side effects; they return values that can be passed to constructors like add.

## Specifying similar constructors

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

```
patterns arith
 is add  | addcc | addx   | addxcc | taddcc
  | sub  | subcc | subx   | subxcc | tsubcc
  | umul | smul  | umulcc | smulcc | mulscc
  | udiv | sdiv  | udivcc | sdivcc
  | save | restore | taddcctv | tsubcctv
constructors
 arith rs1, reg_or_imm, rd
```

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

## Equations

Some instructions have integer operands that cannot be used directly as field values. The most common are PC-relative branches, in which the operand is the target address, but the corresponding field contains the difference between the target address and the program counter. Constructor specifications may include equations that express relationships between operands and fields. Equations contain sums of operand and field values with integer coefficients. For example, the specification for the SPARC branch instruction is

```
constructors
  branch reloc { reloc = $pc + 4 * disp22! }
      is branch & disp22
```

The equation in braces shows the relationship between reloc, the target of the branch, $pc, the program counter, and disp22!, the sign-extended displacement field. The toolkit detects that branch(reloc) is well-defined only if (reloc - $pc) mod 4 = 0, and the generated encoding procedure enforces this restriction. We can introduce a machine-independent notion of program counter without a new specification concept; $pc is simply a predefined identifier that denotes the place in the instruction stream where the constructor emits its tokens.

Some instructions use only part of an operand; for such instructions we can use *slices* of values; for example, val[10:31] denotes the most significant 22 bits of the 32-bit integer value val. This slice is used to specify the SPARC sethi instruction:

```
constructors
 sethi val, rd { imm22 = val[10:31] }
    is sethi & rd & imm22
```

Equations may use inequalities as well as equalities. The toolkit does not use the inequalities to help solve the equations, but it does generate code which checks that the inequalities are satisfied.

## Synthetic instructions and conditional assembly

Applying constructors in pattern definitions is most useful when defining "synthetic" instructions, i.e., instructions that are available in assembly language even though they are not part of the

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

Figure 3: Constructor definitions for the 486's 32-bit addressing modes

real machine. For example, the synthetic instructions `bset` (bit set) and `dec` (decrement) are defined in terms of the real instructions `or` and `sub` (SPARC 1992, p 86):

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

`imode` is needed to convert the integer operand `val` into an operand of type `reg_or_imm`.

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

```
constructors
 set val, rd
   when { val[0:9] = 0 } is sethi(val, rd)
   when { val = val[0:12]! }
                       is or(0, imode(val), rd)
   when { lo = val[0:9] }
     is sethi(val, rd); or(rd, imode(lo), rd)
```

# 7 CISC instructions

All MIPS and SPARC instructions can be specified by conjoining field constraints; this is the property that makes implicit right-hand sides useful. The 486 is not so simple. Both opcode and operands can span several tokens, and some tokens contain parts of each. Fields have multiple uses; for example the field `r_m` can indicate either a register choice or an alternate addressing mode, depending on its value. Figure 3 shows constructor specifications for the 486's addressing modes, illustrating how the toolkit's specification language handles CISC. The brackets and asterisks are not meaningful; they simply help show how the constructors correspond to standard assembly language.

Effective addresses contain a ModR/M byte, which contains an addressing mode and a register. In indexed modes, the ModR/M byte is followed by an SIB byte, which holds index and base registers and a scale factor. Finally, some modes take immediate displacements (Intel 1990, Tables 26-2 to 26-4). We define constructors of type `Eaddr` to create effective addresses in 32-bit mode. The first group of constructors specifies the non-indexed addressing modes. The simplest mode is encoded by `mod = 3`; it is a register-direct mode that can refer to any of the machine's eight general registers. The next three modes are register-indirect modes with no displacement, 8-bit displacement, and 32-bit displacement. The fields `mod` and `r_m` of the ModR/M byte are defined above; the fields `i8` and `i32` occupy full 8-bit and 32-bit tokens and are used to hold displacements:

```
fields of I8   (8) i8  0:7
fields of I32 (32) i32 0:31
```

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

The indexed modes are the second group in Figure 3. The ModR/M byte in which `r_m = 4` is followed by an SIB byte. The stack pointer may not be used as an index register (`index != 4`). Depending on the value of `mod` in the ModR/M byte, the SIB byte may end the address, or an 8-bit or 32-bit displacement may follow. Finally, "`mod = 0 & base = 5`" denotes an indexed address with no base register and a 32-bit displacement.

# 8 Relocation

Instructions often refer to the addresses of data or of other instructions; e.g., to load the value of a variable or to branch to a label. The toolkit can emit such instructions even before the addresses are known. Instructions and data are emitted into *relocatable blocks*, which implement the instruction-stream abstraction. An application can write into a relocatable block without knowing where in memory the block's contents will eventually be located. *Relocation* assigns an address to a relocatable block. Applications may use any number of relocatable blocks.

A *label* points to a location in a relocatable block. The toolkit does not associate names with labels; applications can use any method they want to find and identify labels. A *relocatable address* is any quantity whose value depends on the value of a label. The toolkit's generator treats relocatable addresses as values of an abstract data type. The toolkit's library provides a particular implementation: to the library, a relocatable address is the sum of a label and a signed offset. This simple form is adequate for applications like compilers and linkers. Authors of more sophisticated applications can use more sophisticated representations (e.g., linear expressions over addresses and labels) without changing the toolkit.

Constructor operands whose names begin with `reloc` are relocatable addresses, like the `reloc` operand of the `branch` constructor shown above. The relocatable address `$pc` can be used in a constructor's equations, where it denotes the location at which the constructor's tokens begin. When a constructor that uses relocatable addresses is applied, it checks to see if those addresses are known (i.e., they have been assigned absolute addresses). If so, it treats them as ordinary integers and emits the instruction. Otherwise, it emits *placeholder* tokens and creates a *relocation closure*. The closure contains references to the unknown addresses, plus a pointer to a function that, when applied, overwrites the placeholder with the correct instruction. The application keeps the closure until the addresses it depends on become known, at which point it can apply the closure function and discard the closure.

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

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

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

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

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

# 9 Matching statements

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

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

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

Just as in specifications, it is often more convenient to write patterns in the form of constructor applications, e.g., `fnegs(i, j)`, in which the operands `i` and `j` are bound to integers by the matching statement. Pattern-valued operands are bound not to integers but to locations in the instruction stream; for example, the SPARC pattern `add(0, rand2, rd)` matches any `add` instruction in which `rs1` is zero. `rd` is bound to the number of the destination register, and `rand2` is bound to the location of the token containing the second operand.

Most applications decode streams of instructions in sequence, so they need to find the address of the following instruction as part of decoding the current instruction. The matching statement contains a special syntax for this common operation; if the word `match` is followed by an identifier in square brackets, the matching statement assigns the address of the first unmatched token to that identifier.

The translator implements the matching statement by building a decision tree. Each node of the tree tests one field of one token of the stream being matched. Tokens are identified by width and by offset from the location being matched; for example, the 32-bit displacement in an index-mode add instruction on the 486 might be "the 32-bit token at an offset of 24 bits." The translator first rewrites all patterns so that fields are tagged with the offset of the token to which they belong. It then attempts to build a decision tree that identifies the matching arm using a minimal number of nodes. No polynomial-time algorithm is known, so we use several heuristics. When we examine the trees produced by these heuristics, we see that they are as good as what we would write by hand.

Application writers can use any representation of instruction streams. They specify the representation by supplying the toolkit with four code fragments: the data type used to represent locations, a template used to add an integer offset to a location, a template used to convert a location to a program counter (an unsigned integer), and a template used to fetch a token of a specified width from a location. Widths are measured in bits; offsets are measured in the same units used for the program counter, which defaults to 8 bits per addressing unit. The application writer must supply code that can fetch tokens using the proper byte order, which is usually the byte order of the machine the application runs on.

### Matching statements in `ldb`

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

```
address type  is    "Word.T"
address add   using "Word.Plus(%a, %o)"
address to pc using "%a"
fetch any using "FetchAbs(m, %a, Type.I%w).n"
```

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

Figure 4 shows a simplified version of the SPARC code in `ldb`'s breakpoint implementation, omitting subtleties associated with delayed branches. This code finds which instructions could be executed immediately after an instruction at which a breakpoint has been planted (Ramsey 1994a). After an ordinary instruction, the only instruction that can follow is its inline successor, as computed by the first arm of the matching statement. `FollowSet.T{succ}` is a set of addresses containing the single element `succ`. Calls and unconditional branches also have only one instruction in their "follow set," but conditional branches have two. The two `jmpl` patterns are indirect jumps through registers; the `GetReg` procedure gets the value in the register in order to compute the target address. Using the matching statement implemented by the toolkit makes it clear what the code is doing; the logic would be obscured if implemented by nested case statements.

# 10 Implementation

The toolkit's generator and translator are 6000 lines of Icon (Griswold and Griswold 1990). The

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

Figure 4: Matching statement used for control-flow analysis of SPARC instructions

| Machine | Spec | Ops | Insts | Modes | Time |
|---|---|---|---|---|---|
| MIPS | 127 | 113 | 158 | 1 | 27 |
| SPARC | 193 | 184 | 260 | 2/4 | 102 |
| Intel 486 | 460 | 496 | 623 | 8 | 635 |

| | |
|---|---|
| Spec | Lines of specification |
| Ops | Opcodes in the manual's tables |
| Insts | Instructions specified |
| Modes | Address modes in most instructions |
| Time | Seconds to create encoding procedures (elapsed time on a SPARCstation 10) |

Table 1: Characteristics of three machines

| | MIPS | | | SPARC | | |
|---|---|---|---|---|---|---|
| Program | Bin out | Asm out + | Run `as` | Bin out | Asm out + | Run `as` |
| `eqntott` | 4.1 | 4.3 + | 7.1 | 4.0 | 4.1 + | 2.5 |
| `li` | 7.2 | 7.5 + | 14.7 | 7.9 | 8.2 + | 5.0 |
| `espresso` | 14.7 | 17.5 + | 33.0 | 14.2 | 15.4 + | 11.4 |
| `gcc` | 52.3 | 60.5 + | 233.2 | 60.0 | 61.7 + | 64.7 |

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

Table 2: Seconds to generate code & make `a.out`

library is 600 lines of ANSI C. Table 1 shows some characteristics of our three machine specifications, including the amount of time needed to generate a complete set of encoding procedures. The number of addressing modes affects that time, because each encoding procedure has an alternative for each mode of each operand. Even the long generator time for the 486 is acceptable, because one rarely writes a specification containing hundreds of new constructors. (One can add a few constructors to an existing specification in time proportional to the number of added constructors, not to the size of the whole specification.)

The translator takes 10 seconds to transform either `ldb`'s SPARC follow-set matching statement (Figure 4) or the analogous MIPS statement into Modula-3 code. The translator time, although shorter than the generator times, is more problematic, because the translator must be run after every change to a source file with matching statement.

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

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

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

## 11 Related work

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

Wick (1975) describes a tool that generates assemblers based on descriptions written in a modified form of ISP (Bell and Newell 1971). His work investigates a different part of the design space; his machine descriptions are complex and comprehensive. For example, they describe machine organization (e.g., registers) and instruction semantics as well as instruction encoding. We prefer to build applications by using several simple specifications, each describing different properties of the same machine, to build different parts.

The GNU assembler provides assembly and disassembly for many targets, but different techniques have been applied *ad hoc* to support different architectures (Elsner, Fenlason, et al. 1993). For example, 486 instructions are recognized by hand-written C code, but MIPS instructions are recognized by selecting a mask and a sample from a table, applying the mask to the word in question, then comparing the result against the sample. On both targets, operands are recognized by short programs written for abstract machines, but a different abstract machine is used for each target. Another set of abstract machines is used to encode instructions during assembly. The implementations of the abstract machines contain magic numbers and hand-written bit operations. The programs interpreted by the abstract machines are represented as strings, and they appear to have been written by hand.

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

## 12 Future work

The names of instructions may conflict with names that application writers use in their programs; generating encoding procedures with those same names can cause name-space collisions. Different languages offer different solutions to this problem, e.g., C++ classes or Modula-3 interfaces. Plausible solutions for C include attaching a unique prefix to the names of all encoding procedures, or using a structure containing pointers to the encoding procedures. The latter choice costs more at run time, but could enable multiple sets of encoding procedures in the same application, e.g., one to emit binary and one to emit ASCII (for debugging).

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

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

We are designing an extension to the toolkit that enables it to describe arbitrary sequences. Such sequences are common components of network messages, and this extension would make it possible to use the toolkit to generate encoding and decoding code for such messages. We are also investigating the use of toolkit specifications to help build compression models for arithmetic coding (Bell, Cleary, and Witten 1990). It may be increasingly useful to compress machine code as the gap between processor speeds and secondary storage widens. Better compression would help reduce the storage requirements for large network traces, which can consume gigabytes (Duffy *et al.* 1994).

# 13 Discussion

Our specification language evolved from a simpler language used to recognize RISC instructions in a retargetable debugger (Ramsey 1992, Appendix B). That language had field constraints and patterns built with conjunction and disjunction, but no concatenation and no constructors. There was no notion of instruction stream; instructions were values that fit into a machine word. We extended that language to specify encoding procedures by writing a constructor name and a list of field operands to be conjoined. This scheme sufficed to describe all of the MIPS and most of the SPARC, and we used it to generate encoding procedures for `mld`. There are a few parts of the SPARC it could not describe, however, and it was completely unable to describe the 486, even with the addition of concatenation to the pattern operators. Two changes solved all our problems: making patterns explicit on the right-hand sides of constructor specifications, and using constructor types to permit patterns as operands. We then realized there was no reason to restrict constructors to specifying encoding procedures, so we made it possible to apply constructors both in pattern definitions and in matching statements, yielding the language described in this paper.

Patterns are a simple yet powerful way to describe the binary formats of instructions. Field constraints, conjunction, and concatenation are all found in architecture manuals, and together they can describe any instruction on any of the three machines we have studied. They're not limited to traditional instruction sets in which opcode and operand are clearly separated; all three machines use instruction formats in which opcode bits are scattered throughout the instruction. Disjunction does not make it possible to specify new instructions, but it improves specifications by making it possible to combine descriptions of related instructions. By removing the need to specify each instruction individually, disjunction eliminates a potential source of error.

If patterns provide a good high-level description of binary encodings, constructor specifications raise the level of abstraction to that of assembly language. Equations, though seldom used, are needed to describe instructions like relative branches, whose assembly-level operands differ from their machine-level fields. Equations can also express constraints, which are part of the definitions of some architectures, like the Intel 486.

We maximize the power of the toolkit's specification language by minimizing restrictions on the way patterns, constructors, and equations can be combined. For example, patterns and constructors can be used in each other's definitions, which makes it possible to factor complex architectures like the 486. Equations in constructor specifications are used for both encoding and decoding, and equations can also be used in matching statements. Because the parts of the language work together, it is hard to see how the language could be simplified without destroying it. The only obvious candidate for removal is the conditional-assembly construct, but it is a natural extension of using equations to specify constructors, requiring only a little extra syntax. The simplicity of the specifications and the checking done by the toolkit combine to give users confidence in the correctness of the generated code.


## Acknowledgements

This work has been funded by a Fannie and John Hertz Fellowship, an AT&T PhD Fellowship, an IBM Graduate Research Fellowship, and by Bellcore. More colleagues than we have room to mention were kind enough to review preliminary versions of this paper. We are especially indebted to David Keppel and Karin Petersen for their thorough readings and suggestions.


## Author information


After a fleeting career as a physicist, **Norman Ramsey** (`norman@bellcore.com`) returned to computing in 1986. He likes to build programming tools that people actually use. He received the PhD degree from Princeton University in 1993. **Mary Fernandez** (`mff@cs.princeton.edu`) is a PhD candidate at Princeton University and is an avid user of Norman's programming tools.


*The New Jersey Machine-Code Toolkit is available by anonymous* `ftp` *from* `ftp.cs.princeton.edu` *in directory* `pub/toolkit`. *Its home page on the World-Wide Web is* `http://www.cs.princeton.edu/software/toolkit`.

*This paper was prepared using the* `noweb` *tools for literate programming (Ramsey 1994b). The SPARC examples have been checked for consistency with Appendix A. All of the examples have been extracted from the paper and run through the toolkit, and they all work with version 0.1a.*

# References


Ball, Thomas and James R. Larus. 1992 (January). Optimially profiling and tracing programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM.

Bell, C. G. and A. Newell. 1971. *Computer Structures: Readings and Examples*. New York: McGraw-Hill.

Bell, Timothy C., John G. Cleary, and Ian H. Witten. 1990. *Text Compression*. Englewood Cliffs, NJ: Prentice Hall.

Duffy, Diane E., Allen A. McIntosh, Mark Rosenstein, and Walter Willinger. 1994 (April). Statistical analysis of CCSN/SS7 traffic data from working CCS subnetworks. *IEEE Journal on Selected Areas in Communications*, 12(3):544–551.

Elsner, Dean, Jay Fenlason, et al. 1993 (March). *Using* `as`*: the GNU Assembler*. Free Software Foundation.

Ferguson, David E. 1966. The evolution of the meta-assembly program. *Communications of the ACM*, 9(3):190–193.

Fernandez, Mary. 1994 (November). Simple and effective link-time optimization of Modula-3 programs. Technical Report CS-TR-474-94, Department of Computer Science, Princeton University.

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.

Griswold, Ralph E. and Madge T. Griswold. 1990. *The Icon Programming Language*. Second edition. Englewood Cliffs, NJ: Prentice Hall.

Hastings, Reed and Bob Joyce. 1992 (January). Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136.

Intel Corporation. 1990. *i486 Microprocessor Programmer's Reference Manual*.

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

Nelson, Greg, editor. 1991. *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall.

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

Ramsey, Norman. 1992 (December). *A Retargetable Debugger*. PhD thesis, Princeton University, Department of Computer Science. Also Technical Report CS-TR-403-92.

———. 1994a (January). Correctness of trap-based breakpoint implementations. In *Proceedings of the 21st ACM Symposium on the Principles of Programming Languages*, pages 15–24, Portland, Oregon.

———. 1994b (September). Literate programming simplified. *IEEE Software*, 11(5):97–105.

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

Svrivastava, Amitabh and David W. Wall. 1993. A practical system for intermodule code optimization. *Journal of Programming Languages*, 1:1–18. Also available as WRL Research Report 92/6, December 1992.

Wick, John Dryer. 1975 (December). *Automatic Generation of Assemblers*. PhD thesis, Yale University.

# A Partial SPARC spec

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

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

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

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

```
patterns
 [TABLE_F2 call TABLE_F3 TABLE_F4]
    is op = {0 to 3}
 [unimp _ Bicc _ sethi _ fbfcc cbccc ]
    is TABLE_F2 & op2 = {0 to 7}
```

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

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

The following patterns represent Table F-4 on p 229.

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

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

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

Address operands, defined on p 84, have four possible formats.

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

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

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

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

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

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

```
patterns
 logical is and | andcc | andn | andncc
          | or  | orcc  | orn  | orncc
          | xor | xorcc | xnor | xnorcc
 shift is sll | srl | sra
 arith
 is add  | addcc | addx   | addxcc | taddcc
  | sub  | subcc | subx   | subxcc | tsubcc
  | umul | smul  | umulcc | smulcc | mulscc
  | udiv | sdiv  | udivcc | sdivcc
  | save | restore | taddcctv | tsubcctv
 alu is  arith | logical | shift
```

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

```
constructors
  alu rs1, reg_or_imm, rd
```

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

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

where

  *p* `is any of` [ *a b* ... *z* ],
  `which is` *generating expression*

is syntactic sugar for

  [ *a b* ... *z* ] `is` *generating expression*
  *p* `is` *a* | *b* | ⋯ | *z*

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

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

The assembly syntax for branch instructions is defined on pp. 119-120.

```
constructors
  branch reloc { reloc = $pc + 4 * disp22! }
      is branch & disp22
```

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

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