

Generating a Recognizer from Declarative Machine Descriptions

João Dias and Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
{dias,nr}@eecs.harvard.edu

Abstract

Writing an optimizing back end is hard because it combines three tasks: understanding the target machine, understanding the compiler internals, and writing code. We minimize the intellectual and programming effort by isolating the machine-specific knowledge in declarative machine descriptions. Then, we can analyze the machine descriptions to automatically generate machine-specific components of the back end. In this work, we demonstrate how to use SLED and λ -RTL machine descriptions to generate a recognizer. A recognizer is a fundamental component of instruction selection in several compilers, including vpo and Quick C--.

1 Introduction

In a retargetable compiler, machine-dependent code is encapsulated in the back end of the compiler. Because the compiler's intermediate representation insulates the front end and optimizer from machine-dependent details, only the back end must be replaced to add a new target machine. In particular, a new instruction selector must be written, the register allocator must be reconfigured, the calling conventions must be changed, and machine-specific optimizations may be written. In addition to the great programming effort, writing a back end requires the intellectual effort of understanding the internals of the compiler and the properties of the target machine.

Ideally, a compiler writer should be able to write code for the compiler without worrying about the target machines. Similarly, a machine expert should be able to add a target machine without understanding the internals of the compiler. The first step in this direction is to isolate the knowledge of the machine in a *declarative machine description*.

A declarative machine description clearly and precisely describes a property of a machine.¹ For example, the SLED machine-description language (Ramsey and Fernández, 1997) describes the binary and assembly encodings of machine instructions, and the λ -RTL machine-description language (Ramsey and Davidson, 1998) describes the semantics of machine instructions. Because a declarative machine description encodes knowledge of the target machine in a form that is independent of any client (e.g. a compiler), it is well-suited to formal analysis, and

¹A declarative machine description is not to be confused with the machine descriptions used in compilers such as gcc (Stallman, 1990), in which the machine description is a mapping from the compiler's intermediate representation to machine instructions.

it can be reused by multiple clients. Furthermore, an author of a machine description may have confidence in the accuracy of the machine description if it may be checked for correctness or consistency, as in the case of SLED machine descriptions (Fernández and Ramsey, 1997).

Using formal analysis of machine descriptions, our ultimate goal is to generate the machine-dependent parts of a compiler's back end. By generating components of the back end, we can isolate knowledge of the machine in declarative machine descriptions while significantly reducing the programming effort required to retarget a compiler. In this work, we focus on using declarative machine descriptions to generate code for instruction selection.

In many compilers, instruction selection takes arbitrary intermediate code and finds machine instructions that implement the intermediate code. An alternative approach to code generation is to restrict the intermediate code such that each instruction in the intermediate code can be implemented by a single instruction on the target machine. This restriction is called the *machine invariant*. To support the machine invariant for different targets, the compiler must use an intermediate representation, such as register-transfer lists (RTLs), that is machine independent but capable of representing machine-specific instructions. Because RTLs provide the flexibility to represent instructions on many machines, they can also represent instructions that do not exist on a specific target. Therefore, the machine invariant must be enforced as a dynamic invariant.

The machine invariant is established by a compiler phase called the *expander*, which runs early in the back end. Each subsequent stage of the compiler is responsible for maintaining the machine invariant by using the *recognizer*. The recognizer is a predicate that determines whether an instruction in the intermediate representation can be implemented by an instruction on the target machine. Using the recognizer, compiler phases are written to manipulate RTLs in a machine-independent fashion, then verify that the machine invariant is maintained by calling the recognizer on the resulting RTLs. Any operation that invalidates the machine invariant must be rolled back. For example, the recognizer is called in the inner loop of the peephole optimizer to ensure that the resulting instructions satisfy the machine invariant. The recognizer is also capable of emitting assembly code for machine instructions.

The purpose of maintaining the machine invariant is to support optimization at the level of instructions on the target machine. Because the optimizer is exposed to the semantics of the actual machine instructions, it can take advantage of

instruction features such as complex addressing modes without requiring optimization passes that are specific to particular machines.

In this work, we focus on generating a recognizer from declarative machine descriptions. There are two distinct challenges to generating a recognizer: bridging the semantic gap between a low-level declarative machine description and the abstractions of a compiler, and generating code for an efficient recognizer. We bridge the semantic gap through the use of analyses of the declarative machine descriptions. To generate an efficient recognizer, we construct a tree matcher.

The resulting recognizer has been tested in the Quick C-- compiler. The resulting object file is about 483 KB, which can be compared to about 151 KB for the hand-written recognizer generated using an implementation of iBURG (Fraser et al., 1992a) written in OCaml. We believe significant improvements can still be made to reduce the size of the generated recognizer. The most significant difficulties in integrating the recognizer with the compiler were related to the abstractions in the compiler and the occasionally incorrect encoding of instructions in the hand-written expander. In future work, we will automatically generate the expander as well.

2 The semantic gap

There is a great semantic gap between the model of the machine that is used in a machine description and the model of the machine that is used in a compiler. To explain the source of this semantic gap, we begin with an explanation of the machine model used in λ -RTL.

2.1 The λ -RTL machine model

The λ -RTL machine-description language models the semantics of the machine from the perspective of the processor's execution engine. Each instruction is viewed as a transformation on the state of the machine: the machine is in some arbitrary state before the instruction executes, then the instruction executes and mutates the state of the machine.

The state of the machine is stored in storage spaces on the machine. A storage space on the machine consists of an array of cells in which machine state is stored. For example, on the x86, the storage spaces include the 'r' space for the 32-bit integer registers, the 'm' space for 8-bit addressable memory, and the 'f' space for the floating-point register stack. We refer to a storage location using array-index notation; for example, $\$r[0]$ refers to the first cell in the 'r' space. Because the semantics of the instructions treat registers and memory uniformly as storage locations, there is no need to draw a distinction between different types of storage spaces.

A machine instruction is described according to effects that mutate the state of the machine. An effect represents the process of computing the value of an expression and storing the resulting value in a storage location. For example, using the $:=$ operator to represent the act of storing a value, $\$r[0] := 16 + \$r[0]$ computes the sum of 16 and the value $\$r[0]$, then places the result in storage location $\$r[0]$. This example also makes use of a constant. Because each constant must be encoded as a field in an instruction, a constant in λ -RTL must be expressed as a bit vector (for convenience, the example shows the bit vector in its decimal form).

2.2 A compiler's machine model

The semantic gap arises from the fact that a compiler has a different model of the machine. The compiler manipulates a program at various stages of maturity, including before the *variable placer* replaces variables with pseudo-registers, before register allocation, before stack frames are laid out, and before the assembler and linker. Accordingly, the compiler's model of the machine must address the abstractions used throughout the compilation of a program.

Before the variable placer or register allocator, early forms of a program may include variables or pseudo-registers. The compiler also makes the distinction between registers and memory locations because one of the important tasks of a compiler is to place frequently used values in registers. Accordingly, the compiler's model of the machine must include variables, pseudo-registers, registers, and memory as storage locations.

Before the stack is completely laid out, the compiler may also use late compile-time constants to represent values that are not yet known. Similarly, before the assembler and linker run, the compiler does not know the value of link-time constants. Accordingly, a constant in a compiler may be not only a simple bit vector, but also a late compile-time constant or a link-time constant.

The differences in the representation of locations and constants constitute the semantic gap between the λ -RTL model of the machine and a compiler's model of the machine. In the following sections, we explain the details of the compiler's definition of instructions, as well as the use of λ -RTL and SLED to describe machines. Throughout the presentation of RTLs and machine descriptions, we use the x86 as an example machine, with some simplifications to avoid uninteresting complications while highlighting the important features.

2.3 The compiler's representation of instructions

The compiler represents instructions using a definition of RTLs that matches the compiler's model of the machine. We present the definition of RTLs as a grammar.

The storage locations on the machine are specified in terms of a storage space and the width of the individual cells of the storage space, expressed in bits.

```
space = char
width = int
```

For example, on the x86, the storage spaces include storage space 'r' for the 32-bit integer registers, storage space 'm' for the 8-bit addressable memory, storage space 'f' for the 80-bit registers in the floating-point register stack, and storage space 'c' for the 32-bit control registers (e.g. program counter and condition codes). The compiler also uses storage spaces that do not exist on the machine to represent pseudo-registers.

A storage location may be a register location, a memory location, or a variable:

```
loc = Reg (space, int, width)
      | Mem (space, exp, width)
      | Var (string, width)
```

A register location is accessed by its storage space, an integer index, and a width. A memory location is accessed by its storage space, an address expression, and a width. A variable location is simply described by a name and a width.


```
locations
  EFLAGS is $c[0]
  PC     is $c[1]
```

The `r` storage space represents the 32-bit integer registers, and the `f` storage space represents the floating-point register stack. The number of cells in the `m` storage space is unspecified; therefore, by convention, the space has an infinite number of cells. In practice, the number of addressable memory cells is limited by the size of a pointer. The `c` storage space represents processor state; on the x86, the processor state includes the error flags and the program counter. After the `locations` keyword, the convenient names `EFLAGS` and `PC` are bound to the first and second cells of the `c` space, which is accessed using the `$` symbol and the array subscripts.

Before defining the semantics of the machine instructions, we define the instruction operands. The operands for the x86 include a register index `r`, a 32-bit relocatable constant, a 32-bit memory address, and a 32-bit effective address.

```
operand r      : #3  bits
operand reloc  : #32 bits
operand Mem    : #32 bits
operand Eaddr  : #32 loc
```

An effective-address operand may be either a memory location (constructor `E`) or a register (constructor `Reg`):

```
default attribute of
  E (Mem) : Eaddr is $m[Mem]
  Reg (r) : Eaddr is $r[r]
```

In either case, the effective address refers to a location — either the memory location at address `Mem` or the register `r`.

The `Mem` operand includes addressing modes for indirect addressing through a register and for displacement of an indirect address through a register:

```
Indir (reg)      : Mem is $r[reg]
Disp32 (d32, reg) : Mem is d32 + $r[reg]
```

After the operands have been defined, the instructions can be described in terms of the operands.

On the x86, there are multiple instances of the move instruction, depending on the operands. Two of the instances are defined as follows:

```
default attribute of
  MOV^"mr" (Eaddr, reg) is Eaddr := $r[reg]
  MOV^"rm" (reg, Eaddr) is $r[reg] := Eaddr
```

The semantics of the instructions are defined after the “`is`” keywords in the definition of the move instructions. The `Eaddr` operand provides a form of factoring: as we saw in the definition of the operands, `Eaddr` may stand for either a register or a memory location.

An instruction may define a guarded effect using the infix operator `-->` as in the conditional branch instructions on the x86:

```
Jb.Z (reloc) is x86_z EFLAGS --> EIP := reloc
```

The name of the conditional branch instruction is `Jb`, and the `Z` postfix indicates that the conditional branch is taken on zero. The `x86_z` operator checks whether the conditional branch should be performed; the assignment to the `EIP` register updates the program counter to the target of the conditional branch instruction. In addition to conditional branch

instructions, guards can be used to describe predicated instructions.

The x86’s subtract instruction provides an example of the use of parallel effects, using the infix operator `|` for parallel composition:

```
SUB^mr (Eaddr, reg) is
  let val result is subtract(Eaddr, $r[reg])
  in Eaddr := result
  | set_flags
    {result is result,
     o is sub_overflows (Eaddr, $r[reg])}
  end
```

The second line of the declaration binds the result of the subtraction to the variable `result`, which is used in the body of the `let`-expression. We show only one form of the `subtract` instruction; the other forms differ in the operands and the adjustment of the condition codes. The instruction consists of two effects: the subtraction and the adjustment of the condition codes. The `set_flags` function is defined elsewhere in the specification; `set_flags` precisely models the modifications to the condition codes by setting individual bits of the condition-code register according to the values in the argument record.

The careful reader will note that λ -RTL provides a convenient syntax for specifying instructions as RTLs that are parameterized over operands. For a complete λ -RTL specification of a toy machine invented by Michael and Appel (2000), see Appendix A.

2.5 Describing the binary and assembly encodings of machine instructions

In addition to matching the semantics of machine instructions, the recognizer can emit assembly code or machine code. We specify the binary and assembly encodings of machine instructions using the SLED machine-description language. Because the details of using SLED to define the encodings of instructions are well understood, we present only a high-level overview of the language. Details of the language are described by Ramsey and Fernández (1997), and a sample encoding of Michael and Appel’s toy machine (Michael and Appel, 2000) can be found in Appendix B.

In SLED, the binary encoding of an instruction is described as a sequence of tokens. An instruction on a RISC machine, such as the Sparc, typically uses one token divided into several *fields*. A field is a contiguous sequence of bits within a token. The fields of a token may include the opcode, the operands, and other information.

An instruction on a CISC machine, such as the x86, may include several tokens, including prefixes, opcodes, and operands. Each of these tokens is further divided into fields that specify the exact format of the tokens.

For example, in the SLED description of the x86, the `opcodet` token consists of three fields:

```
fields of opcodet (8) row 4:7 page 3:3 col 0:2
```

This token can be used to define the opcode for a move instruction. The `MOV` pattern specifies the values of the row and page bits of a move instruction, but leaves the `col` field unspecified:

```
patterns MOV is row = 8 & page = 1
```

Because the `col` field helps determine the types of the operands to a move instruction, different values may be used depending on the operands. One possible value for the `col` field is the `Ev.Gv` pattern, which indicates that the instruction may take either 16-bit or 32-bit operands, depending on another token in the instruction:

```
patterns Ev.Gv is col = 1
```

The following constructor defines a move instruction that takes a 32-bit effective address and a 32-bit register as operands:⁴

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

The second line of the definition specifies the name of the instruction `MOVmrod`, which takes an effective address and a register as operands. The third line indicates that the instruction is composed of three tokens (separated by ‘;’). The first token, “`od`”, indicates that the operands refer to 32-bit locations. The second token, “`MOV & Ev.Gv`”, indicates the opcode for the instruction, which is specialized for 32-bit operands by the combination of the “`Ev.Gv`” value and the previous token, “`od`”. The final token, “`Eaddr & reg_opcode = reg`”, specifies the encoding of the effective address and the register in a single token.

The specification of the move instruction has been simplified for the purpose of exposition. The full description of the move instruction on the x86 includes not only the other possible operands to a move instruction but also a concise factoring. Because the encodings of instructions are highly repetitive, even on a CISC architecture, instruction encodings can be factored heavily.

The SLED language also allows the specification of the assembly encoding of instructions. The default printing mechanism is to output constructors (instruction names) and operands according to their given names. This default can be overridden by specifying an alternative syntax for printing constructors and operands. Because constructors and operands are frequently shared between instructions, the specification of the assembly encoding can also be factored concisely.

Although a SLED machine description can not be checked for correctness, it may be checked for consistency (Fernández and Ramsey, 1997). There are two notions of consistency: internal consistency checks for implausible specifications (e.g. missing or overlapping values for a field), and external consistency checks the specification against an assembler.

Among other uses, the SLED tools can generate efficient code to emit instructions as binary or assembly code.

3 Bridging the semantic gap

The semantic gap between the λ -RTL model of the machine and the compiler’s model of the machine manifests itself as a difference in the abstractions necessary to manipulate a program. Several steps during the process of compilation lower the level of abstraction in a program, as seen in Figure 1. From λ -RTL’s perspective, the program has been compiled down to an extremely low level where distinctions between

⁴This presentation has been simplified; the actual machine description factors both the 16-bit and 32-bit instances of the move instruction onto a single line.

storage spaces are unnecessary and constant values are simply bit vectors. But during different phases, the compiler needs a number of high-level abstractions to manipulate a program: variables, registers, late compile-time constants, and more.

To build a recognizer from a machine description, we must construct a function that matches an RTL from the compiler with the semantics of the instructions defined in the λ -RTL machine description. But since the RTLs in the compiler use high-level abstractions, we must raise the instructions specified in the λ -RTL description to the same level of abstraction. For example, storage spaces in a λ -RTL instruction must be replaced by registers or memory, in order to match an RTL generated by the compiler.

However, even after the instructions in the λ -RTL description are raised to the level of abstraction of the compiler’s RTLs, it is possible that two semantically equivalent RTLs may not be syntactically equivalent. Because it is difficult to check semantic equivalence of RTLs (semantic equivalence may not even be computable), we instead compute syntactic equality. Syntactic equality is easily computed, but it requires that the RTLs derived from the λ -RTL description must perfectly match the RTLs used in the compiler. Accordingly, the task of raising the λ -RTL instructions to the abstraction level of the compiler’s RTLs is often combined with the task of establishing conventions for representing instructions as RTLs.⁵ We view the abstractions in the compiler’s RTLs according to three criteria: binding-time abstractions, compiler-phase abstractions, and ad-hoc simplifications.

3.1 Binding-time abstractions

In a λ -RTL machine description, the effects of an instruction are defined in terms of its operands. For example, the x86 has an instruction that adds a sign-extended 8-bit immediate to a 32-bit register. Ignoring the modification of the condition codes, the definition of this instruction in λ -RTL is as follows:

```
ADDIdb (reg, i8) is $r[reg] := $r[reg] + sx i8
```

This instruction adds a sign-extended 8-bit constant to the value in a register. However, a semantically equivalent RTL might add a 32-bit constant that fits in 8 bits to the value in the register.

To ensure that both RTLs would be reduced to the same syntactic representation, we introduce the following principle: any expression that can be evaluated at compile time should be simplified. In our compiler, a component called the *simplifier* is responsible for evaluating any expression that does not depend on a run-time value. The simplifier must be called on an RTL before it is checked by the recognizer.

To match the syntactic representation that the simplifier imposes on the compiler’s RTLs, λ -RTL must apply the same principle. Therefore, instead of attempting to match an RTL with the specification of the `ADDIdb` instruction in the machine description, λ -RTL rewrites the specification to match a 32-bit constant in place of the sign-extended 8-bit constant. However, to preserve the semantics of the instruction, λ -RTL adds a constraint: the value in the 32-bit constant must fit within 8 bits.

⁵These conventions provide no guarantees; they only increase the likelihood that two semantically equivalent RTLs are given the same syntactic representation.

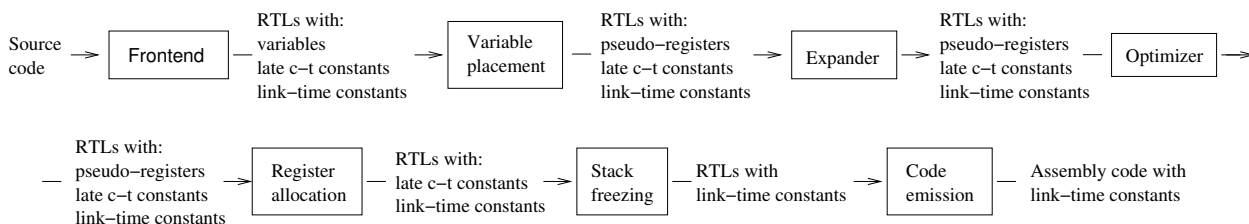


Figure 1: The labeled boxes in the diagram are interesting compiler phases. The text between boxes describes the representation of the program between the adjacent phases of the compiler. The change in the representation caused by the expander is not apparent from the diagram, but before the expander, the RTLs represent arbitrary code; after the expander, the RTLs represent the semantics of instructions that exist on the target machine. One abbreviation is used to describe the RTLs: “late c-t constants” is short for “late compile-time constants.”

Because an instruction in a λ -RTL description is defined in terms of the machine instruction’s operands, the resulting RTLs frequently contain *early* expressions that can be simplified before run-time. To match the RTLs used in a compiler, the λ -RTL toolkit must identify and replace each of the early expressions with simplified expressions. Because the simplified expressions may alter the semantics of the instruction, constraints may be attached to an instruction to ensure that the semantics of the simplified instruction are equivalent to the semantics of the original instruction.

These transformations are based on identifying the *binding time* of an expression. The binding time of an expression is the time at which the value of the expression becomes known. Feigenbaum (2001) classifies the binding time of an expression according to three categories:

- *Specification time*: The value of the expression may be determined from the λ -RTL specification of the instruction because the expression depends only on immediate values.
- *Instruction-creation time*: The value of the expression may only be determined when the instruction is created because the value of the expression depends on the value of an operand.
- *Run time*: The value of the expression is not known until run time because the expression depends on the machine state.

A simple analysis classifies expressions according to their binding times. In λ -RTL, the analysis is used to replace an expression with a constant if the expression’s value can be computed before run time. If necessary, the constant is then constrained by an equation to ensure that the semantics of the replaced expression are preserved.

A similar analysis is used to classify the storage spaces defined in a λ -RTL machine description. Unlike a compiler, λ -RTL does not divide storage spaces into registers and memory. But using an analysis developed by Feigenbaum (2001), we classify a storage space according to the binding time of the expressions used to index into the storage space:

- *Fixed space*: The value of the addressing expression is always known at specification time. An example of a fixed space on the x86 is the register space that contains the program counter and the condition codes.
- *Register-like space*: The value of the addressing expression is always known at instruction-creation time. This criterion for register-like spaces is not a coincidence.

One of the compiler’s most important tasks is to ensure that frequently used values are stored in registers; the compiler can only accomplish this task if it can address the registers at instruction-creation time. An example of a register-like space on the x86 is the integer register space.

- *Memory-like space*: The value of the addressing expression may not be known until run time. An example of a memory-like space on the x86 is the memory space. A more subtle example of a memory-like space on the x86 is the floating-point register stack because it may be indexed according to the run-time value of the floating-point stack-top pointer.

The λ -RTL toolkit uses this classification to identify the different types of storage spaces in a machine description. Using this classification, λ -RTL can match the distinction made between registers and memory in the compiler’s definition of RTLs.

3.2 Compiler-phase abstractions

The compiler’s view of the target machine does not remain constant throughout the phases of the back end, especially with regard to variables and constants (see Figure 1). Although a variable is a valid RTL location, the variables in the program are replaced by pseudo-registers in the variable-placement phase and eventually by registers in the register-allocation phase. Similarly, a late compile-time constant may be used early in the back end, but it is replaced by a bit vector in the stack-freezing phase.

Excluding the optimizer, each phase of the back end in Figure 1 takes one step to strengthen the machine invariant, until the RTLs are ready to be emitted as assembly code. There is a subset relation between stronger and weaker forms of the machine invariant: the machine instructions that appear in the later phases of the compiler are always valid in earlier phases of the compiler, but the converse is not necessarily true.

When the abstract-syntax tree is first converted into the intermediate representation, the RTLs include variables and late compile-time constants. In the next stage, the variable placer replaces the variables with pseudo-registers and memory locations. Then, the expander establishes a weak form of the machine invariant: the RTLs represent specific instructions on the target machine, but they still use pseudo-registers and late compile-time constants. The optimizer then improves the code, maintaining the current form of

the machine invariant. After register allocation, the pseudo-registers are replaced with hardware registers. Then the stack is frozen, and the late compile-time constants are replaced with constant bit vectors. After freezing the stack, the machine invariant on the RTLs has been strengthened to match the assembler’s view of the machine, and the compiler can emit assembly code.

The simplifier also plays an important role in strengthening the machine invariant. After an RTL has been simplified, it no longer contains expressions that can be reduced at compile time.

We would like to use the recognizer at two distinct phases in the compiler: to maintain the machine invariant in the optimizer and to emit assembly code in the code emitter. But the machine invariant evolves between these two phases: register allocation eliminates temporaries, and freezing the stack eliminates late compile-time constants. The simplest solution to this problem is to generate a single recognizer that accepts the weaker form of the machine invariant. When the recognizer is used for code emission, it may match RTLs that contain pseudo-registers or late compile-time constants, but only if the register-allocation and stack-freezing phases are implemented incorrectly. Furthermore, if deemed necessary, the RTLs could easily be checked for the presence of pseudo-registers or late compile-time constants before emitting code.

In the following sections, we focus on how the compiler uses abstractions to account for differences in the representation of instructions in different phases of the back end. In particular, we address the abstractions a compiler may use for variables and constants, and how λ -RTL can raise the semantics of machine instructions to the level of the compiler’s abstractions.

3.2.1 Variables

A variable takes three different forms at different phases of our compiler. When the intermediate representation is first constructed, a variable is represented as a variable in the RTLs. Then, the variable placer replaces variables with pseudo-registers or memory locations. Finally, the register allocator replaces pseudo-registers with hardware registers. Because variables in the RTLs are replaced before the recognizer may be used, we consider only the task of constructing a recognizer that accepts pseudo-registers.

To generate an expander that accepts pseudo-registers, λ -RTL needs to identify the different classes of pseudo-registers and how they may be used in the machine instructions. A pseudo-register must eventually be replaced by one of several registers in a *location set*. We use a location set analysis developed by Feigenbaum (2001) to identify the classes of pseudo-registers:

- *Fixed location set*: A fixed location set contains only a single location. A fixed location set arises if an instruction refers specifically to a location. For example, the 32-bit multiply instruction on the x86 refers to the EAX register; no other location could replace EAX in this instruction.
- *Register-like location set*: A register-like location set contains a set of locations from a register-like space. On the x86, many instructions allow any location in the integer-register space. However, on the Sparc, many instructions accept the subset of integer registers that

```
space = (char)
width = (int)
```

```
const = Bool (bool)
       | Bits (Bits.bits)
       | Late (string, width)
       | Link (Symbol.t, width)
```

```
loc   = Reg (space, int, width)
       | Mem (space, exp, width)
       | Var (string, width)
       | Slice (width, int, loc)
```

```
opr   = (string, width*)
exp   = Const (const)
       | Fetch (loc, width )
       | App (opr, exp*)
```

```
effect = Store (loc, exp, width)
        | Kill (loc)
guarded = (exp, effect)
rtl     = Rtl (guarded*)
```

Figure 2: The RTL type used in the Quick C-- compiler.

excludes $\$r[0]$ because $\$r[0]$ is hardwired to the value 0.

- *Memory-like location set*: A memory-like location set contains a set of locations from a memory-like space. For example, a load from memory on the x86 may load from any memory location.⁶

Because pseudo-registers are used in place of register-like locations, we need one class of pseudo-registers for each register-like location set. A pseudo-register for a register-like location set S can then be used in any instruction that expects a location from S .

This analysis allows λ -RTL to discover how pseudo-registers may be used in instructions. Of course, the compiler and λ -RTL must conspire to use the same names (letters) to represent pseudo-register spaces.⁷

3.2.2 Constants

In λ -RTL instructions, a constant is simply a bit vector. But the assembler provides a number of useful abstractions that are used in a compiler, including labels, constant expressions, and relocatable expressions.

The assembler and linker provide the abstraction of a label, which can be used as an address. Although branch and call instructions may be implemented in hardware as a relative branch from the program counter, the assembly interface provides the illusion that the instructions set the program counter to the value of the label. The difference between the interface provided by the hardware and the interface provided by the assembler requires that the writer of the machine description take care to model the machine at a level of precision that is useful for potential clients. Just as a compiler has no need for intimate details of the processor’s pipeline, it has no need to understand how the assembler and

⁶Of course, the operating system may restrict this set of locations.

⁷Future work may use λ -RTL to generate the variable placer and the definition of storage spaces.

linker conspire to provide labels. Hence, a machine description should describe a jump instruction as provided by the assembler, as demonstrated in the x86 machine description:

```
"JMP.Jb" (reloc) is EIP := reloc
```

The jump instruction takes a relocatable value (e.g. a label) and sets the program counter EIP to the value of the label.

In addition to bit vectors and link-time constants (e.g. labels), the compiler's definition of constants includes late compile-time constants. A late compile-time constant is a constant whose value is not known until the stack-freezing phase of the compiler. Until the value of the constant is known, it is represented by a string and a width, according to the Late variant of constants in the compiler's definition of RTLs (for convenience, the compiler's definition of RTLs is duplicated in Figure 2).

Because the recognizer is called by the optimizer on RTLs that include both late compile-time constants and link-time constants, the recognizer must match RTLs with a more general class of constants than the simple bit vectors in a machine description. The simple solution is to match any compile-time constant where a bit vector is used in the machine description. But unlike a simple bit vector, a compile-time constant may stand for an unknown value. Because the value of the constant may be unknown, the number of bits needed to represent the constant may also be unknown. The danger is that the recognizer may blithely accept the compile-time constant, and when the value of the late compile-time constant becomes known, it may not fit within the required width.

For a machine instruction that uses a constant, such as an instruction that adds an immediate value to a register, the constant must fit within the immediate field of the instruction. If the constant does not fit within the immediate field of the instruction, then the constant must be moved into a register before performing the addition. On a machine that supports variable-sized instructions such as the x86, the size of a constant may not be a limiting factor because the instruction set permits the use of constants that are the same size as registers. However, on a machine like the Sparc or MIPS, the size of constants is an important limiting factor.

There are at least two strategies to handle a constant of unknown width. A pessimistic strategy assumes that the constant must be moved into a register before use. After the stack-freezing phase determines the values of late compile-time constants, a peephole optimization pass can eliminate the use of the register if the value of the late compile-time constant proves to be sufficiently small. Unfortunately, because the register allocator may spill to the stack, it must run before the stack is frozen. Consequently, the register allocator is not free to use the registers that may be freed by the peephole optimization pass.

Alternatively, an optimistic strategy assumes that a late compile-time constant fits in the width required by the instruction. The compiler must be prepared to correct incorrect assumptions after the stack is frozen and the values of the constants become known. On some machines, the assembler may provide support for fixing incorrect assumptions about constant sizes. For example, the MIPS assembler (Kane and Heinrich, 1992) requires that the compiler leave a register unused so that the assembler can copy the values of large constants into the register and use the register in place of the constants.

The assembler may also provide an expression grammar to compute constants and relocatable constants. The proper

way to handle constant expressions is to extend the compiler's definition of a constant to represent any constant expression. Then, wherever the λ -RTL description expects a bit vector, it can match a constant in a compiler RTL, as described in Section 3.1. Similarly, wherever the λ -RTL description expects a relocatable value, it can match a link-time constant in a compiler RTL (Note: a relocatable operand can be identified in a SLED machine description by a simple declaration).

3.3 Ad-hoc simplifications

Several features of instructions may be conceptually simple but require a complicated description to explain all the details. For example, the addition instruction on the x86 may set six different flags in the condition-code register according to the result of the addition. In these cases, compiler writers frequently simplify the representation of instructions by using abstractions.

One useful type of abstraction is the definition of machine-specific operators. Typically, a set of standard operators is defined for use in RTLs. Standard operators may include 2's complement addition and multiplication. However, a machine may perform more complicated operations that can not be expressed conveniently in terms of standard operations. For example, the x86 includes an instruction to compute the tangent of a floating-point number. The simple solution is to add a new operator that represents a pure function that computes the floating-point tangent of its input argument.

Another common abstraction used to represent instructions is the aggregation of mutable state. For example, if an instruction modifies the condition codes, a precise description must update each flag of the condition codes independently. On the x86, many instructions modify the condition codes according to the following λ -RTL function:⁸

```
fun set_flags {result, o, a, c} is
  SF := bit (result < 0) |
  ZF := bit (result = 0) |
  PF := bit (parity (result@bits[0..7]) = 0) |
  OF := o | AF := a | CF := c
```

The `set_flags` function describes six parallel effects to individual, non-contiguous bits of the condition-code register. For the purposes of a compiler, it is sufficient to know that the condition codes are changed according to some operation, without the full detail of how each bit is set.

Sometimes, it is useful to combine the abstractions of machine-specific operators with the aggregation of mutable state. For example, a new operator may be defined to take the two arguments of an addition operation and return the new value for the entire 32-bit condition-code register according to the behavior of the addition operation:

```
rtlop x86_addflags :
  #n bits * #n bits -> #32 bits
```

Then, the modification of the condition codes can be described by a single assignment to the condition-code register:

```
EFLAGS := x86_addflags (x, y)
```

⁸The keen reader will notice that for the purpose of simplifying previous examples, the calls to `set_flags` did not include the `a` or `c` arguments.

In early forms of declarative machine descriptions, the specifications of machine instructions were written out in painstaking detail. For example, the definition of overflow for addition was computed in terms of the resulting value. However, for use in a compiler, it is too much trouble to specify the precise details of each computation. Instead, we use abstractions such as an `add_overflows` operator to simplify the representation of instructions. Of course, these simplifications are another example of semantically equivalent representations that are syntactically different. To ensure that the instructions from the λ -RTL description match the RTLs in the compiler, the λ -RTL description must use the same abstractions to specify the semantics of instructions.

There are a number of advantages of using abstractions to simplify the representation of instructions:

- It is easier to write and understand code that manipulates simpler RTLs.
- The compile-time representation of a simple RTL requires less memory.
- A recognizer that only needs to match simple RTLs may be smaller and faster.

However, simplifying abstractions must be used with care because they may change the semantics of instructions in subtle ways. The aggregation of mutable state may indicate that an instruction uses or modifies more state than it actually does. It is only safe to aggregate mutable state if no source program may detect the difference between instructions with and without this simplification. Because most compilers do not expose the condition codes or the other bits of the condition-code register to a source program, it is generally safe to abstract over mutation of the entire condition-code register. But even when it is safe, a simplifying abstraction may inhibit the optimizer. Because the optimizer may not know the effects of a machine-specific operator or how specific bits of aggregated mutable state are affected, the optimizer may miss out on opportunities to improve the code.

Because simplifying abstractions are a convention used by the compiler writer, they can not be inferred from a machine description. The λ -RTL machine description must be written with the same abstractions as the compiler, as seen in the condition-codes example in this section.

4 Generating a recognizer

The problem of generating a recognizer from a declarative machine description consists of two distinct parts:

1. The analyses from Section 3 are used to raise the abstraction level of the λ -RTL instructions to the level of compiler RTLs.
2. A recognizer that matches a compiler RTL to a machine instruction is generated.

To separate these concerns, λ -RTL constructs generate a specification of the machine instructions in the form of a domain-specific language. The recognizer is constructed by a match compiler that reads the domain-specific language and generates an efficient bottom-up tree matcher in the style of BURG (Fraser et al., 1992b).

4.1 Splitting match generation from match compilation

When the λ -RTL tool reads the λ -RTL and SLED machine descriptions, the semantics of each instruction are represented in a tree structure. The tree structure is almost immediately converted to a form of RTLs that mirrors the abstraction level of the instructions described in λ -RTL: there is no distinction between different types of storage spaces, and the only type of constant is a bit vector. The RTL representation does allow variables, which initially are used to represent the instruction's operands. The RTL representation is well-suited to the analyses that follow.

As described in Section 3, an analysis is performed to determine the binding times of expressions in the RTLs, and each expression that is bound before run time is replaced with a new variable that represents a compile-time constant. To ensure that the semantics of the instructions are preserved, an equation is added to constrain the value of the variable to be equal to the expression it replaced.

A binding-time analysis is also performed to determine which storage spaces represent registers and which storage spaces represent memory (for the purposes of matching compiler RTLs, a fixed storage space represents registers). The RTLs can then be rewritten to replace the references to storage locations with specific references to register or memory locations.

At this point, the RTLs representing instructions from the λ -RTL machine instruction include simplified expressions, compiler constants, and register and memory spaces. The RTLs have now been raised to the abstraction level used in the compiler.

One additional analysis is necessary to facilitate the generation of a tree matcher. In some instructions, an operand may appear multiple times in the semantics of the instruction. To match this instruction, an input RTL must have the same value for each instance of the operand. However, there is no simple way to constrain the instances of an operand during tree matching. Instead, the solution is to linearize the tree such that each instance of the operand is represented by a unique variable, then constrain the instruction such that the values bound to the variables must be equal.

After performing these analyses, the λ -RTL toolkit is prepared to emit the domain-specific language for input to the match compiler. The language is modeled after BURG with the addition of constraints. Like BURG, the input language consists of a set of rules. Each rule has a non-terminal, a pattern tree, a constraint, and an action. The non-terminal is essentially a category: if multiple rules share the same non-terminal, they may be used interchangeably. The pattern tree is the tree that must be matched; it consists of a constructor and a list of sub-patterns. A sub-pattern may be a terminal (string, integer, or char), a non-terminal, or another constructor with sub-patterns. The constraint is an arbitrary string of code that evaluates to a boolean value. The action is an arbitrary string of code that is executed if the rule matches an input tree t . We say that a rule matches t if the rule's pattern tree matches t and the rule's constraint evaluates to true.

For each machine instruction, the λ -RTL toolkit generates a single rule with the `inst` non-terminal (for "instruction"), a pattern tree representing the instruction, a boolean constraint on the variables bound in the pattern tree, and an action that returns assembly code for the instruction. The pattern tree is a simple encoding of the instruction's RTL as

a tree. The boolean constraint is generated by an equation solver, using the constraints gathered to enforce the linearity of the pattern and the semantics of the machine instruction's operands.

The recognizer we currently generate does not match pseudo-registers. In future work, the location-set analysis from Section 3 will determine how pseudo-registers may be used in instructions.

4.2 Techniques for match compilation

The match compiler takes the domain-specific language as input and generates a tree matcher. Tree matching is the task of taking a single input tree (a *subject tree*) and finding a matching tree among a set of *pattern trees*. In the recognizer, the subject tree is the input RTL, and the pattern trees come from the rules in the domain-specific language. We compute only syntactic equality on simplified RTLs.

The basic strategy of a tree-matching algorithm is to traverse the subject tree, visiting each constructor in the tree. A pattern tree matches a subject tree if each non-variable node in a pattern tree matches the corresponding node in the subject tree. A variable node in the pattern tree is considered to match any sub-tree at the corresponding location in the subject tree.

4.2.1 Top-down tree matching

A top-down tree matcher recursively traverses the subject tree from the root node to its leaves. If at any node in the subject tree, the tree matcher discovers that the corresponding node in a pattern tree does not match, then the subject tree can not possibly match the pattern tree.

A top-down tree matcher can be compiled into a decision tree that visits each node of a subject tree only once. But because the decision tree may take different branches depending on the subject tree, the decision tree may require more than one section of code to check a specific node of the subject tree. Therefore, although a decision tree is fast, it may require a large amount of code.

Reasonable heuristics for building top-down matchers are tuned for hand-written patterns with few alternatives and shallow nesting (Scott and Ramsey, 2000). However, a machine's instruction set has hundreds of deeply nested alternatives. Previous work has shown that the space cost of top-down matchers is prohibitive in this setting, even when care is taken to reduce the size of the matcher (Eddy, 2002).

4.2.2 Bottom-up tree matching

A bottom-up tree matcher recursively traverses the subject tree from the leaves back up to the root node. At each node of the subject tree, the tree matcher computes a *state* from the value at the node and the states of the node's sub-trees. If the state at the root of a pattern tree is equal to the state at the root of the subject tree, then the pattern tree matches the subject tree. Each node in the subject tree is visited exactly once while computing the match.

A bottom-up tree matcher can be compiled down to a table-driven matcher. Each table provides the state lookup for nodes that contains a particular constructor, using the sub-states as indices for lookup in the table. The sizes of the tables dictate the size of the tree matcher. Accordingly, table

compression focuses on proving that sub-states are equivalent in some situations (Chase, 1987; Proebsting, 1992). If it can be shown that two sub-states are equivalent as the indices in a table, then the size of the table can be reduced by merging the sub-states into a single index in the table. However, table compression heuristics do not adapt well to poorly factored grammars.

Instead, we found two major ways to decrease the size of the generated code: factoring the pattern trees and factoring the generated code. When λ -RTL manipulates an instruction, it may inline the operands. However, since an operand may be one of several alternatives (e.g. register, displacement from a register, or immediate), inlining requires a copy of the instruction for each alternative choice for each operand. Instead, if we leave the operands in their naturally factored form and generate sub-rules to represent each of the alternatives of an operand, we greatly reduce the size of the generated code. Another useful optimization is to factor the code generated by the match compiler. Although the match compiler may not be able to prove that two states are equivalent, some of the generated code may be shared across states.

The match compiler generates a tree matcher written in OCaml, which is the language in which the Quick C-- compiler is developed. Each table lookup is encoded as a simple pattern match on a flat tuple containing the state numbers of the sub-trees.

4.2.3 Alternative matching algorithms

In addition to tree matching, a recognizer can use other syntactic techniques such as LR-parsing (Aho et al., 1986).⁹ For LR-parsing, the RTL is converted to a string, which is then parsed according to a grammar representing the machine instructions. The main difficulty with this approach is that it is not immediately obvious how to integrate constraints with an LR parser.

4.2.4 Minor complications

There are two complications to the matching problem: variables and constraints. Because we do not know the values of the operands to an instruction specified in λ -RTL, an RTL used to represent a machine instruction includes variables that represent the operands. Because the variables may be referenced in a rule's constraint or in a rule's action, the matcher must bind each variable to the corresponding part of the subject tree. Also, the analyses of the machine descriptions may constrain the RTLs. For example, a variable that represents an operand in an RTL may be constrained to fit within the number of bits required by the machine instruction. Even if a pattern tree matches the subject tree, the rule is not considered a match unless the boolean constraint evaluates to true.

5 Discussion

We have used the λ -RTL toolkit and our match compiler to generate a recognizer for the x86 target in the Quick C-- compiler. Although we started with existing machine descriptions for the x86, we had to make minor modifications to the λ -RTL description to use the same ad-hoc simplifications as our compiler.

⁹Of course, LR parsing is also a form of bottom-up matching.

The major effort of integrating the recognizer with the compiler involved correcting bugs in the expander, which frequently produced the wrong semantics for instructions. For example, the RTL that represented the x86’s block copy instruction was semantically incorrect, but because the hand-written recognizer conspired to recognize the incorrect RTL, the bug had gone undetected. Other bugs in the x86’s expander included missing effects on the floating-point condition codes in floating-point instructions. These types of bugs may be less likely to appear in a machine description because the author of a machine description is free to focus solely on the task of correctly describing the semantics of machine instructions, instead of focusing on getting the compiler to emit the right string of assembly code.

To evaluate the size of our recognizer, we compare the sizes of the stripped object files for three recognizers: a hand-written recognizer, a machine-generated recognizer without factoring the operands, and a machine-generated recognizer with the operands factored out (see Figure 3). The machine-generated recognizer with factored operands

Hand-written using IBurg	150,784 B
Machine-generated, no factoring	987,132 B
Machine-generated, with factoring	483,228 B

Figure 3: The sizes, in bytes, of the recognizers.

is about 3.6 times the size of the hand-written recognizer. A number of factors may contribute to this large gap:

- **Matching strategies:** Our match compiler computes state tables at compile-compile time, whereas IBurg computes states during compile-time. The state tables are well-known to dominate the size of a bottom-up match compiler.
- **Factoring of the input trees:** The IBurg grammar is carefully factored by hand, whereas the input to the machine-generated recognizer is factored only over the operands. In fact, when the machine instructions are not factored at all, the size of the recognizer increases by 70%.
- **Sub-optimal code generation:** Because the match compiler generates the tree matcher in high-level code, it is not always clear what code will look like when it is compiled to assembly. Accordingly, it can be difficult to predict the effects of optimizations made in the high-level language. Among the more effective optimizations have been the limitation of pattern matching to flat tuples and the reduction in the number of source locations that allocate memory. Other opportunities for optimization surely exist, possibly including the reduction of curried functions in the source code.

6 Related Work

Previous work has described how to use declarative descriptions of stack layout and calling conventions to control these components of the compiler (Lindig and Ramsey, 2004; Olinsky et al., 2004). Declarative machine descriptions have also been used for an analysis of register classes (Feigenbaum, 2001) that accurately computes how variables compete for registers (Smith et al., 2004).

We have adopted the compilation strategy of using RTLs, an expander, and a recognizer to represent machine instructions in the intermediate representation. Davidson and Fraser (1984) with *po*, and later Davidson with *vpo* (Benitez and Davidson, 1994), demonstrated that this compilation strategy provides effective machine-level optimization without writing machine-specific compiler passes.

Other researchers have begun to use machine descriptions to generate code generators. Cent et al. (2004) describe a machine-description language for machine semantics that is similar in spirit, if not in syntax, to λ -RTL. Although the full details of the work have not been published yet, it appears that they generate a BURG specification for instruction selection. After the instruction selector, the code proceeds to the register allocator and the instruction selector. This style of code generation foregoes the opportunity to expose the semantics of the machine instructions to the optimizer.

Troeger (2004) used declarative machine descriptions to implement a dynamic binary translation. However, because the effects of an instruction need only be simulated on the host machine, the task is considerably simpler. The machine description maps a single effect to an instruction that executes the effect. Because the execution of the guest machine is only simulated, the effects can be carried out in sequence, as long as the semantics of parallel execution are preserved. Furthermore, the machine invariant is not established on the emulated instructions; rather, the semantic description of the instructions on the host machine is augmented to match any instructions required for binary translation.

The literature on tree matching has been well-developed over the years. Early work by Hoffmann and O’Donnell (1982) on matching trees provides a useful overview of what are now common top-down and bottom-up tree-matching algorithms. Table compression techniques for compile-compile-time state tables in a bottom-up tree matcher were developed by Chase (1987) and refined by Proebsting (1992). An alternative approach to bottom-up parsing is to perform shift-reduce parsing on the intermediate representation (Glanville and Graham, 1978).

7 Conclusion and future work

In this work, we have shown how to use declarative machine descriptions to generate an efficient recognizer for an optimizing back end. Although the generated recognizer is about 3.7 times larger than a hand-written recognizer, it is likely that the size of the recognizer can be reduced through improved code-generation techniques.

In addition to improving the size of the recognizer, future work will include the generation of recognizers for other target machines from machine descriptions. A limiting factor in the generation of recognizers is that the rest of the back end must still be written by hand. Ultimately, we would like to automatically generate more components of the back end, most notably, the machine-specific parts of the expander. Other plans for future work include checking the correctness of a λ -RTL machine description and incorporating semantic comparison to improve the flexibility of the recognizer.

Acknowledgements

Thanks to Paul Govereau and Glenn Holloway for helpful comments on early versions of this paper.

References

- A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.
- Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. In *Programming Languages and System Architectures*, pages 105–124, 1994.
- J. Cent, W. Sheng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. Modeling instruction semantics in adl processor descriptions for c compiler retargeting. In *Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, July 2004.
- D. R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 168–177. ACM Press, 1987. ISBN 0-89791-215-2.
- Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982. ISBN 0-89791-065-6.
- Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, 1984. ISSN 0164-0925.
- Jon Eddy. A continuation-passing operator tree for pattern matching. 2002.
- Lee D. Feigenbaum. Automated translation: Generating a code generator. Harvard College Senior Thesis, 2001.
- Mary F. Fernández and Norman Ramsey. Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering*, pages 326–336, May 1997.
- Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992a.
- Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992b. ISSN 0362-1340.
- R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254. ACM Press, 1978.
- Christoph M. Hoffmann and Michael J. O’Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982. ISSN 0004-5411.
- Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992. ISBN 0-13-590472-2.
- Christian Lindig and Norman Ramsey. Declarative composition of stack frames. In *Compiler Construction, 13th International Conference, CC 2004 Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS CC 2004)*, Barcelona, April 2004.
- Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *CADE-17: 17th International Conference on Automated Deduction*. Springer-Verlag, June 2000. Lecture Notes in Artificial Intelligence, 1831.
- Reuben Olinsky, Christian Lindig, and Norman Ramsey. Staged allocation: Engineering the specification and implementation of procedure calling conventions. 2004.
- Todd A. Proebsting. Simple and efficient BURS table generation. In *SIGPLAN92*, pages 331–340, June 1992.
- Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES’98)*, volume 1474 of *lncs*, pages 172–188. Springer Verlag, June 1998.
- Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.
- Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical report, University of Virginia, 2000.
- Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. ACM Press, 2004. ISBN 1-58113-807-5.
- Richard M. Stallman. Using and porting gnu gcc version 2.0. Technical report, FSF, 1990.
- Jens Troeger. Personal communication, September 2004.

A λ -RTL description of a toy machine

See Figure 4.

B SLED description of a toy machine

On the toy machine language invented by Michael and Appel (2000), the fields of the opcodes and arguments are:

```
fields of instr (16)
  op 12:15 rd 8:11 rs1 4:7 rs2 0:3 c 0:3
```

The opcode `op` can be found in bits 12-15 of an instruction. The possible opcodes for the instructions are specified by their binary encoding. The following syntax assigns the constants 0-6 to the opcodes in the list, in order:

```
patterns [add addi load store jump bgt beq] is
  op 0 to 6
```

Finally, the instructions are specified according to their opcodes and operands.

```
constructors add   rd, rs1, rs2
             addi  rd, rs1, c
             load  rd, rs1, c
             store rd, rs1, c
             jump  rd, rs1, c
             bgt   rd, rs1, c
             beq   rd, rs1, c
```

```

module Toy is
  import [RTL Vector]
  from StdOperators import [ sx --> := | ; = + - < <= > >= ? IEEE754 ]

  storage
    'm' is cells of 16 bits called "memory" aggregate using RTL.AGGB
    'r' is 16 cells of 16 bits called "registers"
    'i' is 1 cells of 16 bits called "processor state"
  locations
    PC is $i[0]

  operand [rd rs1 rs2] : #4 bits
  operand c             : #4 bits

  default attribute of
    add  (rd, rs1, rs2) is $r[rd] := $r[rs1] + $r[rs2]
    addi (rd, rs1, c)  is $r[rd] := $r[rs1] + sx c
    load (rd, rs1, c)  is $r[rd] := $m[$r[rs1] + sx c]
    store (rs1, rs2, c) is $m[$r[rs2] + sx c] := $r[rs1]
    jump  (rd, rs1, c)  is $r[rd] := PC | PC := $r[rs1] + sx c
    bgt   (rs1, rs2, c) is $r[rs1] > $r[rs2] --> PC := PC + sx c
    beq   (rs1, rs2, c) is $r[rs1] = $r[rs2] --> PC := PC + sx c
end

```

Figure 4: A λ -RTL description of the toy machine defined by Michael and Appel (2000).