A Unified Framework for the Automatic Generation of System Tools and Components

ABSTRACT

Machine description languages have long been suggested as a means for automatically generating simulator and compiler tools. Because of the complex nature of computers, designing a language that can describe concisely the intricate details of a machine is difficult. Furthermore, how we describe the machine is often directly influenced by the tools we wish to generate. Assemblers are centered around the syntax of the assembly language, while a decoder for a functional simulator is encoding centric. To generate a suite of related tools we require a language that is generic enough such that we are not restricted to a single tool's point of view. In addition, we need a framework that processes this language into a suitable form that we can then translate easily to generate target tools.

Here we present a framework for building instruction set based tools such as assemblers, disassemblers, and decoders for functional simulators that are automatically generated from the CoGenT instruction specification language, CISL. In particular, we describe the key language features of CISL that enable the construction of an intermediate representation called the *i-graph* and a framework based upon the i-graph that enables modular construction of *translators* for generating tools.

Categories and Subject Descriptors

H.4 [**Compilers and System Tools**]: Miscellaneous

Keywords

Assembler, Disassembler, Architecture Description Language, Mixins

1. INTRODUCTION

Building and extending computer systems requires many tools to enable the construction of new software and hardware. For example, exploring innovative architectural designs requires experimentation with new features using functional and timing simulators. To evaluate these new designs we require a suite of software tools such as compilers, assemblers, disassemblers, linkers, loaders, and debuggers, so that we can generate programs that take advantage of these new design features. Unfortunately, these tools are themselves difficult and time consuming to build by hand.

To alleviate these problems, architectural or machine description languages (ADLs) have been proposed that automatically generate whole tools or parts of tools. This allows architecture-specific details to be written independently from the general design of a particular tool. In addition, ADLs make it easy to modify the specification of the machine for experimentation, and simplifies semantic checking for consistency.

Unfortunately, tools make different demands on the ADL in terms of the required level of granularity and point of view of the machine. For example, circuit level details are supported by some languages [7, 23] for synthesis and layout, whereas others allow one to specify information at the micro-architectural level [10, 13, 24] or instruction set level [19]. Furthermore, the structure of these languages tends to reflect the intended target applications. For example, compiler tools require information for register allocation, relocation, procedure calling conventions, assembly syntax, and instruction encoding, whereas a functional simulator requires details for fetching, decoding, and executing instructions.

To support the diverse requirements of these applications, it is advantageous to provide a *family* of related languages based on similar principles [1]. This allows descriptions to provide the level of detail required by the target tool, but enables information to be shared and extended by other specifications and tools.

Consider the information necessary for generating a timing-accurate simulator, which requires structural information about the micro-architecture components and their interconnections. To use this simulator, however, we need to know how to decode individual instructions. Describing the decode unit at the structural level, however, is not useful in terms of overall timing. Because of this, most timing-oriented languages provide hooks to use a decoder written in an external programming language such as C or C++. A better alternative is to provide hooks that facilitate connection with an instruction set specification language.

Assuming we have a family of coordinated description languages, they are of little use without the compilers and tools that process these descriptions into a form in which it is straightforward to generate target tools. Furthermore, it is equally important that the machine specifications be represented in an intermediate form readily manipulated by tool generators. Tool generators can then use this representation as a database of machine information that they can query for specific details, depending on the tool being generated. This intermediate form is the basis of an extensible *framework* upon which we build tool generators.

In summary, system tools are vitally important in building new architectures for embedded systems. A family of machine description languages allows aspects of a machine to be specified accord-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
.

ing to their level of granularity, as well as for target tools. The structure of the intermediate form of an ADL is key in building the generators that use it to produce target tools. An extensible framework based on this intermediate form facilitates rapid construction of tool generators.

We attempt to address these difficulties by making the following contributions:

- CISL language constructs that facilitate the construction of a unique intermediate representation (the i-graph).
- An algorithm for translating these language features into an i-graph.
- A framework for collecting instruction set information from the i-graph, translating this into tool-specific representations, and generating target specific tools.

We organize the rest of the paper as follows: Section 2 provides a brief overview of the CoGenT project, a family of related descriptions and tools, and its instruction set ADL CISL; Section 3 presents our tool generation framework including the i-graph intermediate form; Section 4 describes applications of our framework to specific target tools; Section 5 describes related work; and Section 6 offers some conclusions and future directions.

2. THE CISL LANGUAGE

The CoGenT project aims to generate tools, particularly compilers and simulators, from a coordinated suite of ADLs. Compilers and simulators for systems research are difficult to develop and coordinate since each tool is complex in its own right, and both are dependent on aspects of the target architecture. The CoGenT project addresses this problem by providing three types of components: multiple coordinated specification languages used for describing various aspects of a computer architecture, including instruction set and micro-architecture; a framework for building translators that process these machine descriptions and generate tool components; and a framework for building complete tools (such as an assembler) from the translator output. CoGenT allows researchers to explore innovative ideas quickly, by expressing new concepts in domain specific languages and automatically generating the necessary tools. Furthermore, it provides a consistent framework for extending the set of translators, and thus the set of tools, that it can generate. In the following sections, we focus on CISL, the CoGenT instruction set language, and how we generate tools from these descriptions.

Instruction set architectures come in varying degrees of complexity. For example, the MIPS architecture [11] has a uniform encoding and straightforward semantics. The SPARC [22] also has a simple encoding, but the semantics are complicated by register windows and branch delay slots. The x86 [6] is even more complicated, with its variable length encoding and CISC style semantics. And the PowerPC [9] and ARM [21] fall somewhere in the middle. To describe accurately current and next generation architectures we require a specification language with constructs for decomposing instructions both by their representation (i.e., encoding, assembly syntax, etc.) and semantics (what they do). In addition, the language must allow all properties of an instruction to be defined naturally and flexibly enough to allow extensions for future features. Lastly, the language must provide mechanisms to encourage description reuse.

CISL is a strongly typed, polymorphic, class-based instruction set description language with a Java-like syntax. It focuses on providing language features for expressing instruction concepts at the bit level and structures for managing their decomposition. CISL provides three primary language constructs for describing instructions: *classes*, *mixins*, and *attributes*. Attributes define individual properties of instructions; classes, along with single inheritance, define the decomposition of an instruction set; and mixins are used to specify characteristics that do not easily fall under the single inheritance of classes. We do not cover all the details of CISL syntax and semantics here.[1] Rather, we focus on the features and strategies that we use to generate system tools.

2.1 Attributes

An *attribute* in CISL looks very much like a method declaration in conventional object-oriented programming languages such as Java and C#. It has a name and an optional return type, and possibly some formal parameters. The body of an attribute method is either empty or contains a list of statements describing something interesting about an instruction. Consider the following attribute definitions from the PowerPC *add* instruction:

```
fun effect() {
  GPR[RT] = GPR[RA] + GPR[RB];
}

fun string syntax() {
  return "add%{A:oo_syntax}%{A:cc_syntax}
          r%{I:RT}, r%{I:RA}, r%{I:RB}"
}
```

The *effect* attribute describes the semantics of this instruction. It performs addition on the values contained in registers `GPR[RA]` and `GPR[RB]` and stores the result in register `GPR[RT]`. We discuss the reference to the register file `GPR` and the instruction fields `RT`, `RA`, and `RB` in Section 2.1.1. The second attribute defines assembly syntax for the instruction using our syntax description language. We discuss the details of assembly syntax in our discussion of assembler generation in Section 4.1.

A critical property of an attribute as a language construct, and a main design goal of CISL, is its generality. The semantics of an attribute and its relevance are deliberately left to be interpreted by the translator tool that processes it. This allows one to add new instruction set attributes when designing a new tool. For example, when processing a CISL description to generate the semantic functions for a functional simulator, the simulator generator looks for the *effect* attribute. Other tools may ignore this attribute allowing descriptions that need not be complete (an important feature for instruction set design). The only common analysis that is required for all generator tools is that the description is recognized as a valid CISL file syntactically. Other analysis, such as type checking semantic descriptions is taken care of by a translator.

Attributes may also be *generic*. That is, the formal parameter types may be left unspecified or partially specified. For example, the following attribute definition in our CISL PowerPC description specifies the semantics for calculating the effective address for the "next instruction address" of a branch instruction:

```
fun setNIA(signed imm) {
  NIA = calcEA( (imm :: 0b00)! );
}
```

Its formal parameter is partially specified as being `signed` (but given no size in bits). A call to this attribute will be valid only if the type of the actual argument *at least* matches `signed`. The rest of the type information is inferred based on the parameter at the call site. (One way of thinking of this is that attribute calls are always fully inlined, which is not a problem since CISL does not need, and therefore does not allow, recursion.)

[1] The final paper will direct readers to related publications.

2.1.1 The Store

The CISL examples given so far refer to pre-defined register files (such as GPR). In CISL a separate description is given for all the memories accessible to instructions in the system. This description is known as *the store* and contains CISL type information for user registers, system memory, and special registers (e.g., the processor status register or PC). These memories are accessed using a syntax similar to array dereferencing in Java.

2.2 Classes

Although attributes are sufficient for specifying the important features of instructions, how we organize those attribute definitions is even more important. CISL provides a *class* construct for grouping together related attributes. Inheritance facilitates concise descriptions of instruction fields and their decomposition. Consider the instruction formats found in the PowerPC. CISL class definitions readily capture these formats and their encodings, as shown in the following complete definition of the PowerPC *addi* instruction:

```
class Instruction {
  var bit[32] inst;
  var bit[6]  OPCD @ inst[0];}

class DForm extends Instruction {
  var bit[ 5] field1 @ inst[ 6];
  var bit[ 5] RA     @ inst[11];
  var bit[16] field3 @ inst[16];}

class DForm_RT_SI extends DForm {
  var bit[ 5] RT @ field1[0];
  var bit[16] SI @ field3[0];}

instruction class addi extends DForm_RT_SI {
  fun encode() { OPCD = 0x0E; }

  fun syntax() {
    return "addi r%{I:RT}, R%{I:RA}, %{I:SI}";
  }

  fun effect() {
    if (RA == 0) GPR[RT] =           SI!;
    else         GPR[RT] = GPR[RA] + SI!;}
}
```

A CISL class that does not extend another class is a *root* class. In the PowerPC ISA, all instructions are 32 bits long and have a 6-bit opcode field. Therefore, our PowerPC description has a single root class called `Instruction` containing three variable (field) declarations.

A CISL variable is either a *concrete variable* or a *variable reference*. A concrete variable declares a single bit or an array of bits with type modifiers such as *signed/unsigned* or *big/little* (endian), whereas a variable reference refers to a concrete variable or another variable reference. In the code above, the `Instruction` class declares a concrete variable called `inst` that is 32 bits long, and a variable reference `OPCD` that defines the 6-bit opcode field starting at bit 0 of the `inst` bit array. These two kinds of variables allow encodings of instructions to be defined in a top-down fashion. Concrete variables are declared to indicate the shape of the instruction word and references are used to refer to fields. The distinction made between concrete and reference variables provides important information to the translator tools (Section 4).

One completes the definition of an instruction by indicating that its class is an *instruction class* (as in the `addi` class above). The inheritance hierarchy up from an instruction class to a root class defines one unique instruction and all of its properties. Although the above example presents all the attribute definitions in an instruction class, that is not required. It is possible (and common) to have attributes that occur at various points in the inheritance tree and that are used in subclasses by "calling" the attribute (as mentioned in Section 2.1). In this case, we kept the example simple to be illustrative and compact.

2.3 Mixins

Although classes are sufficient for describing much of the information related to instructions, we often encounter situations where properties are shared in a manner that is not easily captured by single inheritance. Consider the *shifter operand* in an ARM [21] data-processing instruction. There are 11 different forms represented by this addressing mode, used by 16 different instructions, leading to a total of 176 unique instruction formats and semantics (not to mention the 16 different condition field opcodes). To define the format for these instructions using only single inheritance would require 11 classes for each of the shifter operands and 16 classes that represent each of the data processing instructions, duplicated 11 times, inheriting a different shifter operand class with each. This quickly leads to an unwieldy description that is hard to read, understand, and maintain. What we require is the notion of attaching variables and attributes to classes without disrupting the class hierarchy. In CISL we accomplish this task using *mixins*.

A mixin is similar in content to a class but is *used* by classes rather than extended. When a class uses a mixin, the declarations in the mixin become available to that class. In addition, it is possible (and common) for a class to use more than one mixin. This does not mean, however, that all the declarations in the used mixins become available simultaneously. For example, a class declaration K that uses mixins X and Y actually refers to two unique class definitions: a class K that uses mixin X and a class K that uses mixin Y. In effect, a mixin can be viewed as a kind of *class constructor*. The following shows how we use mixins to describe the encoding for a SPARC *ldsb* instruction with two different encodings for the *i* field:

```
mixin reg_access {
  var bit i @ inst[13];
}

mixin direct_address extends reg_access {
  fun encode() { i = 0; }
}

mixin offset_access extends reg_access {
  fun encode() { i = 1; }
}

instruction class LDSB extends Format3
      uses direct_address, offset_access {
  fun encode() { op3 = 0b001001; }
}
```

We first define the mixin `reg_access` that provides the variable reference declaration for the `i` field (which incidentally resides at offset 13 in the instruction word `inst`). This mixin is then inherited by two sub-mixins: `direct_address` and `offset_access`. These two mixins define different encoding values for the `i` field (0 and 1 respectively). The `LDSB` instruction class then uses *both* mixins for two unique (non-conflicting) encodings defined over the *i* field.

CISL descriptions are concise and explicit, giving the programmer control over formatting and semantics necessarily at the bit level. However, a tool chain to process and analyze these descriptions is at least as important as a legible description.

3. FRAMEWORK

We now discuss the framework depicted in Figure 1, which supports a number of operations. Namely: translation of a CISL description into an intermediate form called an *i-graph*; a reusable

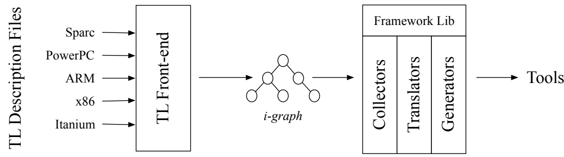


Figure 1: Framework

Algorithm 1: Builds an *i-graph*
Input: C—The class graph
MIXINROOTS—The set of root mixin nodes
Output: I—The i-graph
BUILD-IGRAPH(C,*MIXINROOTS*)
(1) M : Map from classes to a set of classes
(2) I : Graph
(3) U : Set
(4) **foreach** $m \in$ MIXINROOT
(5) Class c' = new class
(6) **foreach** m_i in path $m \rightarrow^* u$
(7) $\mathrm{D}[c'] := \mathrm{D}[c'] \cup \mathrm{D}[m_i]$
(8) $\mathrm{D}[c'] := \mathrm{D}[c'] \cup \mathrm{D}[u]$
(9) $\mathrm{V[I]} := \mathrm{V[I]} \cup \{c'\}$
(10) $\mathrm{M}[c] := \mathrm{M}[u] \cup \{c'\}$
(11) $\mathrm{U} := \mathrm{U} \cup u$
(12) **foreach** $c \in (\mathrm{V[C]} - \mathrm{U}) \wedge \neg\mathrm{MIXIN}(c)$
(13) $\mathrm{V[I]} := \mathrm{V[I]} \cup \{c\}$
(14) **if** $(c,s) \in \mathrm{E[C]} \wedge \mathrm{M}[s] \neq \{\}$
(15) **foreach** $s' \in \mathrm{M}[s]$
(16) $\mathrm{E[I]} := \mathrm{E[I]} \cup (c,s')$
(17) **else if** $(s,c) \in \mathrm{E[C]} \wedge \mathrm{M}[s] \neq \{\}$
(18) **foreach** $s' \in \mathrm{M}[s]$
(19) $\mathrm{E[I]} := \mathrm{E[I]} \cup (s',c)$
(20) **else**
(21) $\mathrm{E[I]} := \mathrm{E[I]} \cup (c,s)$
(22) **return** I

library of strategies we call *collectors* for traversing the i-graph harvesting information; a library of *translators* used to create abstract representations of target tools (e.g., assemblers); and a set of *generators* for producing code for a specific tool suite.

3.1 The CISL Front-end

The first step towards generating target specific tools is parsing a CISL description. The front-end allows specifications to *import* other description files via an `import <name>;` statement. Following the parsing and abstract syntax tree construction of a CISL description, the front-end generates a *working set* of classes and mixins. Because architectures are often extended across successive generations to include such things as larger register files and new instructions, one can redefine or *override* classes and mixins. The working set represents a specific architecture generation.

Given a working set, the front-end then performs standard semantic checks (not including type checking). After the front-end performs its checking phase it proceeds to generate the instruction graph.

3.2 Instruction Graph

After the front-end parses the input files it has an abstract syntax tree representation of the machine specification. This representation, however, is not necessarily the best form for processing and generating tools. For example, we are most often interested in the encoding of a complete instruction. The AST representation has this information, but in general it is distributed across several classes and mixins. The situation is similar for the attributes describing instruction semantics as well as assembly language syntax. On the other hand, the tree representation of the class hierarchy is useful for generating instruction decoders for functional simulators and disassemblers. This, however, is complicated when a class uses a mixin.

Hence, we desire a representation incorporating the contents of each mixin into the classes that use it. In doing so, we keep the structure of the class hierarchy for such things as decoding, while at the same time providing a mechanism for retrieving complete instruction information. We call this representation an instruction graph or *i-graph*. The i-graph is a graph whose vertices are the classes of the CISL description and whose edges represent the inheritance relationships between those classes.

Before we construct the i-graph, however, we begin by creating a graph C whose nodes are the classes and mixins from the AST and whose edges are the inheritance and uses (mixin) relationships. In particular, we include only those classes that are either an instruction class or a class that is extended from an instruction class. In addition, we maintain a set of mixins, *MIXINROOTS*, whose elements are those mixins that are extended by other mixins but do not themselves inherit from anything. The class graph C and the *MIXINROOTS* set are used as inputs to `BUILD-IGRAPH` as outlined in Algorithm 1 and depicted in Figure 2(A).

`BUILD-IGRAPH` first declares a map M that it uses to map classes to a set of classes, the new graph I, and the set U that it uses to remember the set of classes we have encountered that use mixins. The first phase of the algorithm is to construct new classes from classes that use mixins and to add the new members to the i-graph. This is accomplished by the loop over *MIXINROOTS* in lines 4–11. Because mixins have their own inheritance structure, we start with the root mixin and follow the path through its submixins adding mixin declarations, $D[m_i]$, to the declarations of the new class $D[c']$. The loop terminates when we encounter a class u that uses the mixin. We then add the declarations of u to our new class c', which we subsequently add as a new node to the i-graph (I). We place an entry in M that maps the class u to the set of classes that have been generated from it in the i-graph. This will prove important for adding edges. Lastly, we add the class u to the set U of classes that use mixins. Because a mixin may be used by multiple classes, we actually perform the loop over the mixin inheritance paths for each leaf class. For the sake of clarity we omitted this detail from Algorithm 1. The results of this transformation are illustrated in Figure 2(B).

After we add the classes that incorporate the mixin declarations, we add those classes that do not use mixins to the i-graph. That is, those classes that are in $C - U$ (excluding the mixins as t hey are no longer useful). As we do this, we add the relevant edges to the i-graph according to the following three rules for each class c:

- If we have an edge from class c to class s in C (c extends a class s) and we have a mapping from s to classes in I (i.e., $\mathrm{M}[s] \neq \{\}$) we add an edge in I from c to each of the classes s' in I;
- otherwise, if we have an edge from class s to class c in C (c is extended by a class s) and we have mappings to classes in I, we add an edge in I from each of the classes s' to c;
- otherwise, we add an edge from c to s in I.

These rules take into consideration the fact that a class that uses multiple mixins in the original class graph may have multiple instantiations in the i-graph. Because these original classes are never

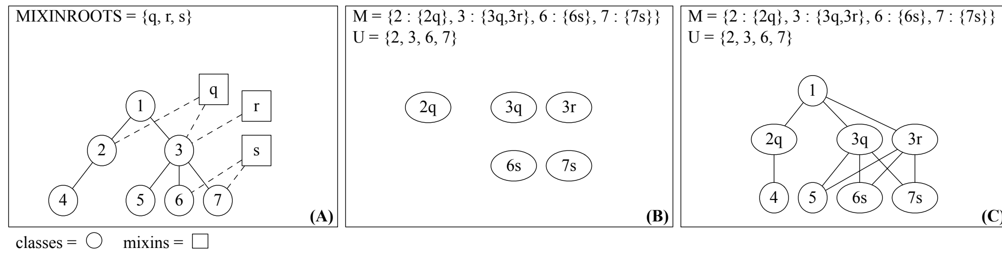


Figure 2: *i-graph* Construction

added to the i-graph directly, we must use the mappings to add the edges to those classes that were generated from it. The resulting i-graph now contains a set of vertices that are either classes from the original graph or new classes that we generated from the mixins and edges that represent the original inheritance relationship. Most importantly, any unique path $r \rightarrow^* i$ in the i-graph, where r is a root class and i is an instruction class (or leaf), represents a unique instruction in the target instruction set architecture. Figure 2(C) shows the i-graph generated from this last transformation. In succeeding sections we discuss that part of the framework that uses the i-graph as a database for collecting application specific information, the translation of that information, and the generation of target specific code.

3.3 Collectors

After the CISL front-end generates an i-graph, we can start harvesting the information that we care about. To facilitate easy construction of generator tools and code re-use, we provide a library of *collectors* that visit the i-graph in various ways. These collectors can then be composed to gather information that is relevant for a particular target tool. For example, a functional simulator requires information related to the field encodings of an instruction, in what order those fields should be decoded, and the instruction semantics. An assembler requires encoding information but also needs details related to the syntax of assembly instructions as well as how symbols in the syntax are bound to the instruction fields in the encoding.

Each collector considers a single instruction at a time. In the context of an i-graph, this is *a single path from a root vertex to a leaf*. This path, however, can be traveled in several different ways using a particular *visiting strategy*. We can start at the root and work *top-down* to the leaf, we can start at the leaf and visit vertices *bottom-up* until we reach a root node, or we can go top-down followed by bottom-up in a single pass, or vice versa. How we visit the vertices of the instruction paths depends on the type of information we are gathering and how we will use it to generate a target component.

Our framework initially defines several general collectors for such things as gathering variables, variable references, and attributes. These general collectors are extended to provide filtering capabilities, such as collecting attributes having a particular name or gathering attributes that are related in some way. For example, our assembler generator tool expects a unique `syntax` attribute to be defined for each instruction. This attribute, however, may reference other attributes of a given name that further define the syntax. As such, we require a collector that only collects attributes that are related to the specification of assembly syntax. We found this collector *pattern* to occur frequently so we provide a general collector for this task that can be re-used by several tool generator applications. These general collectors are then combined along with a visiting strategy to form a *multi-collector* that collects all the information relevant to a specific tool in a single pass over the i-graph. These multi-collectors are also extensible to allow the collected information to be refined further to specialize the presentation of that information to be used by a translator as depicted in Figure 3.

3.4 Translators

Once a collector has gathered the tool specific information, a *translator* is used to transform that information into a representation that is useful for a specific tool suite. In other words, as a collector provides a particular view of the information it finds in the i-graph, a translator presents a view of that information in the context of a specific target tool. Our overall approach here is not much different than the phases used in compiler construction. The source programming language is translated into an AST and then subsequently into a tuple or tree-like IR language (our collector phase). This IR is then converted into a lower-level form that closely resembles the target instructions but may still be independent of the output format (our translator phase).

Thus, the job of a translator is two-fold: translate the information provided by the collectors into a form that is suitable for a particular target tool suite; and perform any semantic checking on that information to ensure correctness. Because the information provided by a collector is target-independent, the translator must use that information to construct a representation that is in the context of the particular application. For example, we may have two target simulator frameworks, for which we wish to generate decoders. Each may have their own peculiarities such as how they interface to the rest of the simulator and whether or not they are side-effecting a simulated store or calculating timing. In effect, the translator is the "glue" that binds the machine-specific information to the target-specific tool.

In many cases, we require the output from a collector to satisfy certain properties or conditions. For example, for encoding and decoding applications we must be confident that the fields defined for a particular instruction do not overlap and that the encodings are disjoint (i.e., no two instructions have identical encodings) and for functional simulators, in particular, we require that the semantic effect of an instruction has valid semantics and types and we may want to perform analysis and optimizations over the decode tree. As we mentioned previously, the meaning of the attributes and their contents are dictated by the translators. This allows different translators to apply semantics differently depending on the target application.

Similar to collectors, translators can be composed in a pipeline fashion to operate over the collector output. This facilitates the sharing of translators that may be useful for two different tool generators that target the same tool suite or to provide a general trans-

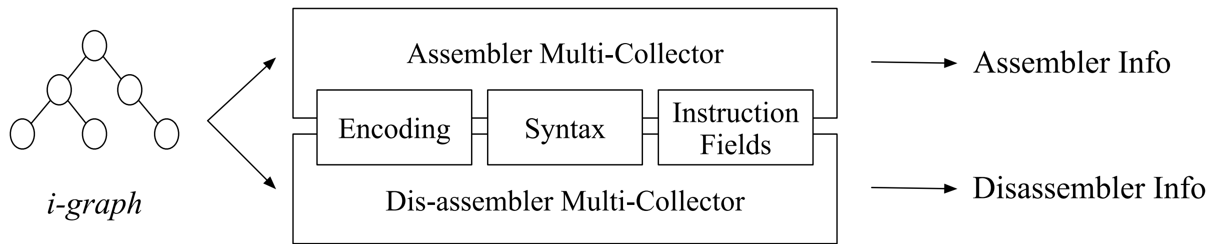


Figure 3: Using and Re-using Collectors

formation that can be used by several generator tools such as common semantic checks.

3.5 Generators

After the instruction set information is processed by translators, their output representation flows to a *generator* to generate code in the particular programming language for the target tool. Consider an emitter module for a C compiler containing "emit" functions for generating encoded instructions for a target machine. If the C compiler is written in C, we would build a generator that would take the output of an emitter translator and generate the machine specific module in C. On the other hand, if the C compiler is written in Java, we would generate Java code with a different generator.

Besides generating code for a target programming language, a generator could just as easily generate input for another processing tool. This allows us to easily integrate CISL and it's framework into a legacy tool generation scheme, to be utilized by tools written in other languages, or even generate instruction set documentation to be viewable in a web browser. The entire framework and information flow is illustrated in Figure 4.

4. APPLICATIONS

In this section we demonstrate the effectiveness of our framework with specific application examples. In particular, we show how the CISL language and tool generator framework is used together to generate important components for an assembler and disassembler. We conclude this section by mentioning other tools that could benefit from using our approach.

The following examples will illustrate the flexibility of the Collector, Translator, and Generator pattern. In each example, a set of collectors sweeps the i-graph. Translators operate on this information, generating abstract operations (meta-ops) representing procedures in a language and platform independent way. Lastly, generators process meta-op trees and output platform-specific code in a given language. This multi-phase approach gives CISL the ability to specify assembler and disassembler construction in an abstract way, and to generate assemblers and disassemblers for new systems by simply writing new generators.

4.1 Generating Assemblers

The basic job of an *assembler* is to read in a file containing a program for the target machine and generate a binary object file. To accomplish this task, the assembler needs to know the syntax of the assembly language and the encoding of its instructions. Most assembly files also contain sections for program text and data as well as special syntax for indicating the types of data, assembler hints, important locations (such as the address of the current assembly instruction), labels, and possibly more. An assembler usually supports a rich notion of symbols, expressions, forward references, name resolution, synthetic instructions (i.e., `set` instruction on the SPARC and `mr` instruction on the PowerPC), and back-patching. We focus here on generating the core part of an assembler i.e., parsing assembly instructions and generating their binary encoding.

An assembler needs to extract three categories of information from a CISL description: instruction syntax, encoding, and synthetics (i.e., rewrite rules). We require the syntax of an instruction in order to parse the assembly from a text file, the encoding for emitting the binary representation, and the synthetics for specifying alternate assembly syntax. First, we discuss how we specify assembly syntax and synthetic instructions.

As mentioned previously, the collectors and translators dictate the semantics of the attributes and their contents. Our assembler collector expects each instruction to define a *syntax* attribute. This attribute contains a *format string* representing the syntax of the instruction. A format string contains literal characters and *directives* for describing how variable fields are either read (in the case of an assembler) or written (in the case of a disassembler). In addition, these directives describe the binding of symbolic names in the assembly syntax to fields in the instruction encoding. This allows a complete format string to be used for both mapping syntax to encoding and encoding to syntax.

4.1.1 Format Strings and Directives

The format of directives is: "%{*directive-name:arguments*}", where *directive-name* is an identifier and the *arguments* is a optional comma separated list of arguments beginning with a colon. Consider the syntax for a register on a typical RISC machine. It usually begins with an 'r' followed by the integer number of the register. This is accomplished using the *I* directive:

```
fun string syntax() {
  return "r%{I:rd}"
}
```

For an assembler, we parse the character 'r' followed by an integer whose value is stored in instruction field `rd`. Alternatively, from the standpoint of a disassembler, we generate the string containing 'r' followed by the integer `rd`.

Additionally, we have specific directives for decimal, hexadecimal, octal, and binary as well as more involved constructs such as labels (*L*), conditionals (*C*), calls to attributes (*A*), and hooks (*H*) for calling external code in the assembler application. Assemblers usually require a back-patching pass to resolve label references, so we created a special directive for labels to indicate their purpose, such as storing mappings from label symbols to instruction offsets as well as building a standard symbol table (useful for disassembling). Here is an example of the SPARC *call* instruction using labels:

```
fun syntax() {
  return "call %{L:disp30}";
}
```

An assembler treats a label as an entry into a map from strings to addresses and postpones binding the `disp30` field until the first

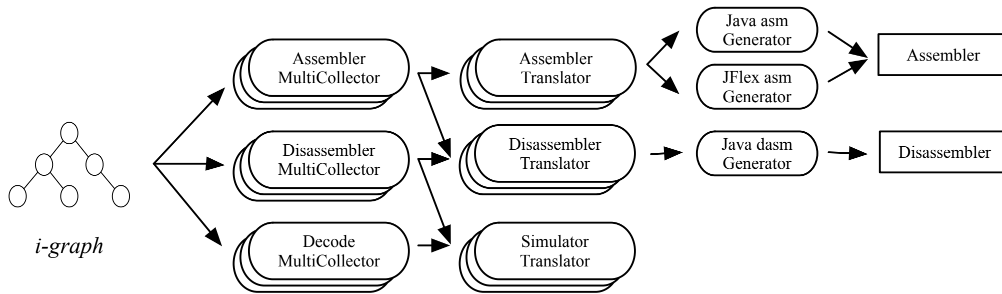


Figure 4: Framework Flow

pass is done. A disassembler uses the value of the `disp30` field to either generate or lookup a symbol (in a symbol table, for instance), then printing this symbol rather than just the raw address.

Assembly syntax typically accepts several variations of a single instruction. For instance, when specifying if condition codes should be set (the 'o' and '.' syntax for the PowerPC) or whether an instruction is conditionally executed (e.g. the ARM). Other cases include addressing modes that can be used with an instruction. We use the *C* directive to specify this "conditional" syntax. This is illustrated in the SPARC *ldsb* instruction:

```
fun string syntax() {
   return "ldsb [%r%{I:rs1} %{C:(simm13 == 0)(),
        (simm13 < 0)(- %{I:simm13}),
        (?)(+ %{I:simm13})}], %r%{I:rd}";
}
```

A conditional directive has arguments of the form (*cond*)(*format*) where *cond* is a boolean expression on a field and *format* is a format string. For the example above, the string is conditional on the `simm13` instruction field. If the simm13 field equals 0, the format is empty; if simm13 is negative, the format string is "-%{I:simm13}"; otherwise (indicated by `?`) the format string is "+%{I:simm13}".

The *A* directive is used to "invoke" the attribute given as its argument. The target attribute must have a return type of *string*. This string is substituted in the format string at the attribute directive.

4.1.2 Synthetic Instructions

We must also be able to describe the syntax of synthetic instructions. A synthetic instruction is an alternate syntax for a real instruction or a list of real instructions. In CISL, synthetic instructions are specified using the *synthetic* keyword as a modifier on a class:

```
synthetic class cmpd {
  fun syntax() {
    return "cmpd %{I:bf}, %{I:ra}, %{I:rb}";
  }
  fun equivalent() {
    return "cmp %{I:bf}, 1, %{I:ra}, %{I:rb}";
  }
}
```

A synthetic class contains two attributes: the `syntax` attribute for its own syntax and the `equivalent` attribute which describes the effect in terms of non-synthetic instructions.

4.1.3 Collector Support

To generate assembler components you need syntax, encoding, and synthetic information. These three collectors extend the *NamedAttributeCollector*, which searches the i-graph looking for attributes with a given name. The *SyntaxCollector* visits each class node in a path and looks for attributes named `syntax` that return a `string`. Because the syntax attribute can be declared at any point along the instruction path (as can attributes it depends on), the collector must be able to visit the i-graph in a bottom-up or top-down fashion. The collector produces a list of attributes corresponding to the syntax of a single instruction.

The *EncodingCollector* is the composition of two generic collectors: a NamedAttributeCollector and a *VariableCollector*, used to gather all the variables declared along a single path. The NamedAttributeCollector collects all attributes named *encode*, which assign encoding information to fields of the instruction. Its body consists of a list of assignment statements called *constraints*. The left-hand side of a constraint is either a real variable or a variable reference and the right hand side is an integer. The result of this collector is the complete format of an instruction and all its known encoding information. Those fields without encoding information are set by encoders (i.e., assemblers) or extracted by decoders (i.e., disassemblers, functional simulators).

The *SyntheticCollector* collects the `syntax` and `equivalent` attributes from synthetic classes. Since these classes are not contained in the i-graph we need to search the initial working set for all synthetic classes. We then map from the format string in the syntax attribute to the format string in the equivalent attribute (to generate a rewrite rule).

All three collectors are composed to create the *AssemblerCollector*. This information is stored in a list of *meta-instructions* where each meta-instruction contains all the information gathered from each of the collectors mentioned above. After we build the list of meta-instructions, translators are used to convert that information into a form that can be used to generate an assembler. The syntax of an assembly instruction will generate a lexer for parsing an assembly file.

4.1.4 The Syntax Translator

Given the list of format strings collected for each instruction, the SyntaxTranslator expands all attribute calls within the syntax format string. This proceeds until we reach a string without attribute directives, which is parsed into a list of literal strings and format directives.

Instruction encoding mandates the declaration of at least one concrete variable. Subsequent reference variables can always be mapped onto a concrete variable declarations (perhaps through several layers of reference). These field references form a hierarchy of types that define a function for accessing bits in the underlying concrete variable.

4.1.5 Assembler Meta-Ops

An assembler must manipulate information at the bit level to encode instructions. Because CoGenT is meant to generate tools for many different languages and systems, we must be able to map our

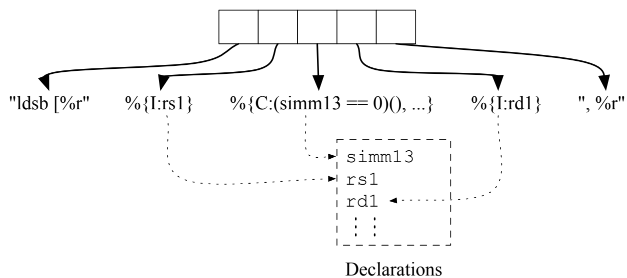


Figure 5: Instruction Assembly Form

abstract assembler operations onto language specific operations and constructs. For example, contrast C and Java. C hides very little of the underlying machine, while Java insists on specific endianness, signedness and bit widths for its data types. Even though both languages have bit shift and masking operations C on a 64-bit little endian machine would require very different code from Java. The translator converts collected information into a language-independent representation of *meta-ops*. A meta-op provides a limited set of operations that allow us to manipulate bits in an abstract manner. This describes location, access patterns, and type of a particular field. These meta-ops are input to a target specific generator that emits target code for each meta-op.

Figure 7 shows the translation of the BF field (a 3-bit field contained in a PowerPC DForm instruction) into meta-ops. On the left, we show the bit representation of a PowerPC instruction word. To the right, are the CISL declarations for the fields. The instruction word, `inst`, is a big unsigned 32 bit array; `field1` is a 5 bit reference starting at bit 6 in `inst` (note that indexing starts from the left as these variables are big-endian). `BF` is a 3 bit reference on `field1` at bit 0.

Each variable reference is translated into meta-ops. This translation outputs a series of *shift* and *mask* meta-ops determined from the size and endianness of the field. The endianness dictates whether we generate a right or left shift, and the size determines the shift size and mask value. The type hierarchy is traversed one level at a time, building a tree of shift and mask meta-ops. The final meta-op tree can be used directly or as a candidate for further processing and optimization.

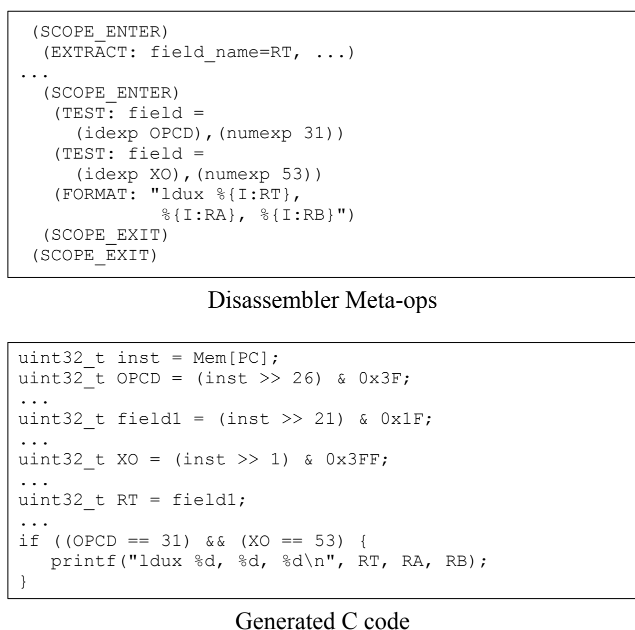
After the assembler translators have compiled the collector information, the assembler generators use this to generate code in the target programming language. For example, generating a lexer file with rules derived from the meta-op trees.

4.2 Generating Disassemblers

Disassemblers map binary encodings of instructions to assembler syntax. Generating disassemblers is not much different than generating assemblers—the major difference is generating the decoder tree. Using our framework, we leverage the assembler collectors with an additional collector: *DecoderTreeCollector*. This collector builds a tree from the class inheritance hierarchy. The tree nodes hold *decoder meta-ops* that describe the fields of an instruction. Note that the decoder generated for disassembly can also be used (albeit with a different generator) as the decoding logic for a simulator.

4.2.1 Disassembler Meta-Ops

The disassembler is constructed from five categories of meta-ops. The *extract* meta-op, generated from CISL field references, describes how values are extracted from fields in much the same way



```
 (SCOPE_ENTER)
  (EXTRACT: field_name=RT, ...)
...
  (SCOPE_ENTER)
   (TEST: field =
     (idexp OPCD),(numexp 31))
   (TEST: field =
     (idexp XO),(numexp 53))
   (FORMAT: "ldux %{I:RT},
            %{I:RA}, %{I:RB}")
  (SCOPE_EXIT)
 (SCOPE_EXIT)
```

Disassembler Meta-ops

```
uint32_t inst = Mem[PC];
uint32_t OPCD = (inst >> 26) & 0x3F;
...
uint32_t field1 = (inst >> 21) & 0x1F;
...
uint32_t XO = (inst >> 1) & 0x3FF;
...
uint32_t RT = field1;
...
if ((OPCD == 31) && (XO == 53) {
   printf("ldux %d, %d, %d\n", RT, RA, RB);
}
```

Generated C code

Figure 6: Disassembler Collection and Translation

that *shift* and *mask* meta-ops describe similar functionality in the assembler. The *fetch* meta-op indicates that new bits must be fetched from memory, and is the result of concrete field declarations. Proper processing of these meta-ops ensures that the description does *not* need to provide explicit figures for the number of fetched bits used in the decode process. While this is a trivial matter with RISC ISAs, it is extremely helpful for variable-length prefix-driven CISC ISAs (such as Intel's IA-32). *test* examines encoded constants and compares them with existing fields, *format* describes the output syntax for a particular (fully-resolved) instruction, and the *scope* meta-ops allow subsequent translator and generator passes to infer the structure and scope of the other meta-ops.

Figure 6 shows the generated meta-ops from the decoder tree of a subset of the PowerPC instruction set. We have omitted the original CISL code due to space constraints. First, the value of `RT` is extracted (through the reference to `field1` from the instruction word. Then, if `OPCD` is 31 and `XO` 53, the instruction is an `ldux` with the provided format.

4.2.2 Disassembler Translation and Generation

Translation and generation of the disassembler is straightforward. Since the *DecoderTreeCollector* has already determined the control structure of the resulting decode tree (based on how the information is collected and how the *scope* meta-ops are positioned), the only remaining task is to optimize this tree and translate into the implementation language.

4.3 Current Results and Further Applications

We currently have complete descriptions for the SPARC and PowerPC, as well as partial ARM and AMD-64 descriptions. Our assembler and disassembler generators are producing meta-op trees, and we are in the process of implementing generators to produce the stand-alone versions of these tools. We expect most of the software and descriptions currently under development to be available for download from the project web page in the near future.

As mentioned in the disassembler discussion, the decoder can be used to generate decoding logic for a functional simulator. It is also

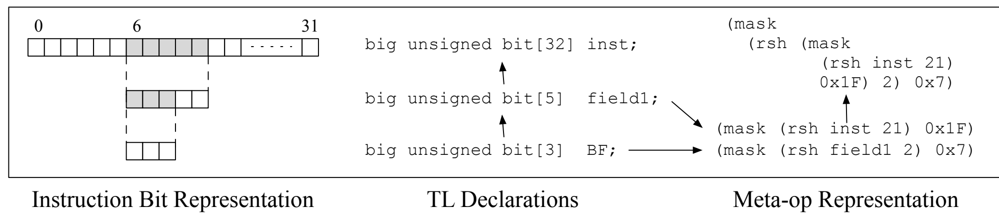


Figure 7: Variable to Meta-op Translation

possible to see how Collectors gathering `effect` attributes would be used to generate instruction semantic functions for a simulator. Code emitter libraries can also be generated from the extracted encoding and syntax information.

5. RELATED WORK

CISL is not the first description language invented for ISAs. Early description languages [2] were often tied to a specific architecture family. Later languages (such as nML [3], SLED [16, 17, 19, 18], λ-RTL [15], and MLRISC [4]) overcame this particular limitation. nML was designed for machine synthesis, however and was unsuitable for integration in multiple compiler tool chains (a large goal of the CoGenT project). SLED is an instruction encoding description and has to be combined with λ-RTL in order to describe instruction semantics. This partitioning, in addition to an overly terse syntax causes the λ-RTL + SLED system to be unwieldy. MLRISC was designed almost completely for compiler construction, and so was very attractive, however it has a fixed input representation (MLTREE) which would require the compiler writer to implement conversion routines between their IR and MLTREE. This conflicted with the CoGenT project's goal of seamlessly integrating with existing compiler tool suites. In addition, the compiler focus of MLRISC made it a poor match for the CoGenT goal of generating cycle-accurate timing simulators. Project Maxwell [8] automatically generates several system tools (assemblers, disassemblers) for several different architectures, but it requires the specifications to be written in Java. This is a problem because Java cannot express certain kinds of operations concisely (e.g. sign-extension, rotation, or population count). Additionally, the system lacks support for simulator components.

Architecture description languages (ADLs) such as Liberty [24], LISA [13], EXPRESSION [5], and MADL [14] are focused on simulator generation and synthesis and require internal extensions or calls to external code to perform compiler or decoding tasks. ADLs have also enjoyed recent popularity in the ASIP community, where the goal is to generate a set of ISA extensions to greatly speed the execution of a specific family of applications. LISA [20, 12] has been used to generate simulators and system tools (assemblers and debuggers). The Tensilica corporation [25] has developed a description system to develop simulators and compiler tools for ASIP design. However, their tool-generation system is proprietary.

The dream of being able to generate accurate simulators and efficient compiler and system tools from the same set of descriptions is an old one. Many tools have been created over the years, but because the problem is quite difficult many tools and languages sacrificed generality in the face of ISA complexity. Many systems are either heavily compiler-biased or simulator-biased. Some are biased towards a specific family of architectures, or require a very specific input IR in addition to a machine description. Because the CoGenT project aims to be a general tool suite for compiler and simulator generation, these more highly focused languages could not be used. CISL is a general machine description language, that nonetheless can be used to generate tools useful for both compilation and simulation.

6. CONCLUSION AND FUTURE WORK

All low-level system tools, including compilers, simulators, assemblers, and disassemblers require detailed information concerning the target architecture. Therefore, it is sensible to use a single description to generate all of these components. This paper has presented a new framework based on the CISL description language for generating components regardless of the implementation language. Specifically, we described an algorithm for generating a graph-based intermediate representation, as well as a three phase process for converting that representation into language-specific utilities. We also have provided a detailed description of how to leverage the features of our language and this framework to generate some commonly-required system tools.

As our system matures, we plan to generate functional simulators in a variety of languages, components for compiler back-ends (such as instruction schedulers and code generators), and other standalone system tools (such as debuggers).

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant number XXX-0000000. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] M. W. Bailey. *CSDL: Reusable Computing Ssytem Descriptions for Retargetable Systems Software*. PhD thesis, 2000. Advisor-Jack W. Davidson.

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

[3] A. Fauth, J. V. Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[4] L. George and A. Leung. MLRISC: A framework for retargetable and optimizing compiler backends. Technical report, Bell Laboratories and New York University, 2000.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM Press.

[6] Intel Corporation, http://www.intel.com/products/processor/manuals/index.htm. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2007.

[7] R. Lipsett, C. F. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

[8] B. Mathiske, D. Simon, and D. Ungar. The project maxwell assembler system. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 3–12, New York, NY, USA, 2006. ACM Press.

[9] C. May, E. Silha, R. Simpson, H. Warren, and I. CORPORATE International Business Machines, editors. *The PowerPC architecture: a specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[10] C. W. Milner and J. Davidson. Quick piping: a fast, high-level model for describing processor pipelines. In *Proceedings of the Joint Conference on Languages Compilers and Tools for Embedded Systems (LCTES)*, pages 175–184, 2002.

[11] MIPS Technologies Inc., http://www.mips.org. *MIPS Technologies*, 2007.

[12] S. Pees, A. Hoffmann, and H. Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):815–834, 2000.

[13] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA—machine description language for cycle-accurate models of programmable DSP architectures. In *Design Automation Conference*, pages 933–938, 1999.

[14] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 47–56, New York, NY, USA, 2004. ACM Press.

[15] N. Ramsey. Using an ml-like language to specify the semantics of machine instructions.

[16] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*, pages 172–188, June 1998. Available as Springer Verlag LNCS 1474.

[17] N. Ramsey and M. F. Fernandez. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA, Jan. 1995.

[18] N. Ramsey and M. F. Fernandez. Automatic checking of instruction specifications. In *1997 International Conference on Software Engineering*, pages 326–336, May 1997.

[19] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[20] O. Schliebusch, A. Chattopadhyay, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, and T. Kogel. A framework for automated and optimized asip implementation supporting multiple hardware description languages. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 280–285, New York, NY, USA, 2005. ACM Press.

[21] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[22] SPARC International Inc., http://www.sparc.com. *The SPARC Architecture Manual*, 2007.

[23] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1995.

[24] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty structural specification language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.

[25] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 184–188, New York, NY, USA, 2001. ACM Press.