International Journal of Parallel Programming, Vol. 33, Nos. 2/3, June 2005 (© 2005)
DOI: 10.1007/s10766-005-3587-1

CISL: A Class-Based Machine Description Language for Co-Generation of Compilers and Simulators

J. Eliot B. Moss,[1] Trek Palmer,[1] Timothy Richards,[1] Edward K. Walters, II,[1] and Charles C. Weems[1]

It is currently difficult fully to understand the performance of a modern dynamic programming language system, such as Java. One must observe execution in the context of specific architectures in order to evaluate the effects of optimizations. To do this we require simulators and compiler back-ends for a wide variety of machines that are capable of handling the demands of today's dynamically compiled languages and their environments. We introduce here CISL, a machine description language specifically designed for the automatic generation of simulators and compiler back-end. CISL is a class-based language with a C/Java style syntax aimed at extensibility. CISL is processed by tools to generate descriptions of architectures represented in an intermediate form; the descriptions are then further combined and processed to produce efficient compiler and simulator components designed to "plug in" to existing frameworks. CISL provides the necessary flexibility to advance the simulation paradigm to match the state of the art in computer systems.

KEY WORDS: Language design; machine description; simulator; compiler; instruction set architecture.

1. INTRODUCTION

Compiler and architecture performance analysis requires a highly accurate simulation infrastructure to obtain useful results. It is important that we

[1]Department of Computer Science, University of Massachusetts Amherst, Amherst, MA 01003-9264, USA. E-mail: {moss, trekp, richards, ekw, weems}@cs.umass.edu

0885-7458/05/0600-0231/0 © 2005 Springer Science+Business Media, Inc.

make observations using a compiler that is capable of producing optimizations specialized to the simulated target. Both simulator construction and compiler back-end construction are time-consuming and error-prone. These issues seriously hamper efforts at architectural exploration and compiler optimization work. Also, it is difficult to obtain simulators with the right level of detail necessary for experiments involving modem dynamic languages, such as Java.

An integrated simulator/compiler environment, driven by higher-level specifications, is needed to advance the simulation paradigm to match the state of the art in computer systems. We propose a system, called Co-GenT, that will automatically generate matching compiler back-ends and efficient functional and cycle-based simulators. Functional simulation and code emitters will be generated directly from provided machine descriptions. Timed simulators and instruction schedulers will also be generated from the descriptions augmented by timing models.[1,2] The glue necessary to turn a code-emitter into a compiler's code-generator will be derived automatically from descriptions using bounded heuristic search.[3]

The CoGenT system is a collection of tools and libraries for manipulating and transforming machine descriptions, and as such the first major step in constructing CoGenT is the creation of a common language for those in descriptions. CISL, the CoGenT Instruction Set Language, is a class-based language with a C/Java style syntax. CISL aims at extensibility and modularity. In CISL, the class hierarchy and mixin functions can be leveraged to provide clear and succinct descriptions of instruction syntax and semantics. CISL provides the foundation for an adaptable system for generating matching compiler back-ends and simulators.

This paper provides an overview of CISL. Section 2 presents some of the prior work upon which the project is based. Section 3 contains an overview of some of the more interesting features of CISL. Section 4 provides an explanation of how CISL is used within the framework of the larger CoGenT project. Section 5 explores some of the upcoming topics that will be addressed by this research, and Section 6 concludes.

2. PRIOR WORK

Machine description languages have been a topic of research for some time. Two fairly recent languages are SLED[4] and λ-RTL.[5] These languages were created separately, and although they have different syntax, they were combined to form the basis of a confederation of languages for the generation of compiler back-end tools. The focus of SLED was to specify the syntax of machine instructions whereas λ-RTL was designed to specify instruction semantics.

These languages had the feature that there were separate definitions for instruction syntax and semantics. The disadvantage of this approach, however, is that it ignores the inherent coupling of syntax and semantics and makes the matching of the two the responsibility of an additional tool. This matching makes the construction of tools that use these languages complicated and difficult to implement. Furthermore, the multi-language approach complicates the transcription of an architecture manual into a complete machine description. In addition, the languages over-emphasize a shorthand-like conciseness which has a major impact on readability. The readability and usability issues have made the SLED/λ-RTL confederation difficult to employ in practice. It is because of these short-comings we felt it necessary to create CISL.

An earlier language, nML,[6] combined the description for both instruction syntax and semantics. It described instructions using an attributed grammar and thus exploits the hierarchical nature of most instruction sets allowing the sharing of common structures. However, the syntax of the nML language is insufficiently expressive to be convenient.

There are many other examples of machine description languages, such as ADL,[7] EXPRESSION,[8] Facile,[9] and MLRISC,[10] but these languages are either too low-level for our purposes, require the use of specialized environments, or were designed for use with a specific tool set. In hardware design, VHDL[11] and Verilog[12] are the most common hardware description languages, but we do not require the level of detail that they provide.

Cattell's thesis[3] is an important work in automatic derivation of code generators. Cattell built a code generator generator (CGG) that, given a machine description of a target, would build a CG for that target, to be incorporated into a specific compiler using a specific (but reasonably generic) intermediate representation (IR). Cattell's CGG works by performing a bounded heuristic search, for each IR instruction, of all possible sequences of target instructions that have the same net semantics, using semantics of the IR and of the target expressed as algebraic trees. Cattell's CGG worked quite well and fairly quickly even by today's standards.

As successful as this achievement was, it had several drawbacks. The IR that Cattell was using was itself not specified but rather hard-coded into the CGG, whereas we desire to work from any given IR by starting with a description of that IR, as well as of the target ISA. In addition, the IR Cattell was using used the same architectural store description as the target architecture. This greatly simplifies the matching process and does not support matching from instruction sets operating on different machine stores. Cattell's descriptions are also not modular. Drawbacks aside, the

core of Cattell's work is valid and effective, and the design of CISL has been influenced by the desire to make heavy use of Cattell's ideas.

Machine description languages also exist for purposes other than describing ISA syntax and semantics. To generate accurate cycle-based simulators automatically, one needs either a micro-architectural description language[2] or an abstract method of describing timing.[1] A current topic of our research is the construction of such a language compatible and consistent with the CISL design philosophy.

3. AN OVERVIEW OF CISL

CISL is a domain-specific language designed to describe patterns of bits and computation over those patterns. Each pattern represents bits in their own abstract space, which may or may not be disjoint with the spaces of other patterns. The real machine counterparts of these abstract spaces are the different stores of the machine, such as registers or memory. Specifications written in CISL represent patterns in the form of classes, which contain data members and semantic methods.

This section provides an overview of some of the interesting and innovative features of CISL, and as such it is not a complete description. A more thorough presentation appears in the CISL manual.[13]

3.1. Data Primitives

3.1.1. Bits and bit arrays

Any possible representation of information by a conventional binary architecture can be reduced to a pattern of bits, so the basic element of any CISL description is a *bit array*. A bit array is an ordered set of bit values. Individual bits are accessed by giving the integer representing their index (0-based) in the array, surrounded by brackets. A bit array variable of size 32 with the name *inst* is declared as:

```
bit [32] inst;
```

and bit 31 within *inst* can be accessed by

```
inst [31];
```

CISL also supports multidimensional bit arrays. The following declares a bit array representing a register file containing 32, 32-bit registers:

```
bit[32,32] regs;
```

and bit 5 of register 4 is accessed by

```
regs[4,5];
```

In addition to reading and writing individual bits, bit arrays can be accessed and mutated as a whole. The assignment of one bit array to another is legal as long as the dimensions of both arrays are the same, i.e., the number of dimensions and the size of each dimension matches. For example, the following assignment is valid subject to these definitions:

```
bit[30, 29] uneven;
bit[30, 29] uneven2;
bit[29, 30] uneven3;
bit[30, 30] even;

uneven = uneven2;
```

But the following are invalid:

```
uneven = even;
even = uneven;
uneven = uneven3;
```

Note that even though *uneven* and *uneven3* contain the same number of bits, their dimensions are not the same and therefore one cannot be assigned to the other.

3.1.2. Bit array references

Bit Array References allow one to treat a range of bits within a previously defined array as a bit array in its own right. References are most often used to represent a sub-range within a larger bit array that we wish to manipulate independently. Note that bit array references are not copies—they access the actual bits represented by the reference, so mutation of the reference will mutate the bits of the larger array. An informal way of thinking of bit array references is that they provide an *overlay* of additional structure on top of a larger bit array. This has proven to be a useful way to view references, so *overlay* is provided as an optional keyword (see Section 3.2.2). For example, if the *inst* variable declared above represented an instruction for a RISC architecture, then the first six bits of the array might represent an opcode value (call it *op*) in the RISC ISA. The following will define a bit array reference of size 6 beginning at index 15 of *inst*:

```
bit [6] op @ inst[15];
```

3.2. Types

3.2.1. Type declarations

For CISL to be powerful enough to describe non-trivial patterns, it must be possible to construct more complex types from the simple

bit arrays described above. We do this using *type declarations*. Type declarations allow one to declare types that may then be referenced in the construction of more complex types. For example, if within a specific architecture we want to consider each register within a register file to be an unsigned 32-bit integer, we use the following type declaration:

```
type uint = big unsigned bit[32];
uint[32] regs;
```

In terms of basic bit arrays, this defines a bit array of 32 rows, each row of which is 32-bits in size, is indexed in a big-endian manner, and represents an unsigned value.

3.2.2. Type modifiers

Type modifiers are prepended to an array variable declaration or reference to modify the interpretation of the bits contained within the array. Modifiers can dictate the direction of indexing of a bit array (*bigendian* or *littleendian*), whether the abstract value represented by the array has sign information (*signed* or *unsigned*), and whether or not bit array references are allowed to be made to a bit array (*overlay* or *final*). Note that type modifiers do not change the actual pattern of bits within the array, only the meaning of the array within a larger context. It is possible to preserve the abstract value of a bit array but change the bit pattern (e.g., changing endianness).

Type modifiers may be specified any number of times in a particular declaration or over nested declarations (see Section 3.2.3). Conflicting type specifiers, such as *little big bit*[32] are also allowed, with the convention that the leftmost modifier in a declaration has precedence. CISL parser will report the conflicts so the description writer is aware of them.

For multi-dimensional arrays, it is possible to have a different set of type modifiers for each index. This is accomplished by declaring the array elements independently, and combining them using a type declaration.

Endianness, determines the direction of indexing on a bit array. A bit array declared as little-endian (keyword little) assigns an index of zero to the least significant element of the array. A bit array declared as big-endian (keyword big) assigns an index of zero to the most significant element of the array. Compared to a little-endian indexing scheme, the big endian scheme accesses the same bits (or array elements) by starting at a array index *size-1* and decreasing to zero.

Arrays declared within arrays can have differing endianness. This is accomplished by declaring the arrays with a different endianness and combining the types. For instance, a little-endian array of 32-bit big-endian values (declared in this context as *uint32*) would be declared thus:

```
little uint32 [1024] memory;
```

Note that this does not refer to an array of 1024 *unit 32* elements whose endianness has been converted to little-endian. Instead, this is an array that is indexed in a little-endian fashion, and each element in the array is in turn a bit array indexed in a big-endian fashion.

As a further example, this multi-dimensional array declaration describes the pattern representing memory as seen by the Intel family of processors. Memory consists of an array of words accessed as little-endian, each word consists of four bytes accessed as little-endian, and each byte is a size 8 bit array accessed as big-endian.

```
big bit[8] byte;
little byte[4] word;
little word[1024] memory;
```

Converting from one endianness to another without altering the bit pattern is possible through straight assignment from a variable with one endianness to one with the opposite endianness. The resulting array, when viewed from the original endianness, will appear to have a different abstract value.

Sign, CISL admits two kinds of sign modifiers: `signed` and `unsigned`. The purpose of these modifiers is to determine how sign-sensitive operators should interpret the abstract value of a bit pattern. Sign modifiers also may have effects such as sign extension if a bit array of smaller size is coerced into one of a larger size.

3.2.3. Subranges

Subranges allow one to refer to sub-portions of a type. Subranges are to types as references are to bit arrays. Subranges are also analogous to union types in C, where the same space can be interpreted according to different representations. Here is an example of a type declaration followed by a subrange declaration based upon that type:

```
type uint = bit[32];
subrange fb = bit[1] @ uint[0];
```

The subrange declaration specifies the size of the new type in bits as given by *bit* [l], and the start location of the subrange within its containing type (given by *uint*[0]).

Subranges can subsequently be used in references in the same manner as C unions. For example:

```
uint rl;
rl.fb = 1;
```

Here, a variable *r*1 of type *uint* is declared, and the bit corresponding to subrange *fb* is set to 1.

Subranges of subranges are allowed, but recursive subranges are forbidden. It is also possible to have a subrange with different type modifiers from the type that is being subranged. Here is an example based upon the IEEE 32-bit Floating Point format:

```
type ieee32FP = big unsigned bit [32];
subrange sign = bit [1] @ ieee32FP [0];
subrange exp = bit [8] @ ieee32FP [1];
subrange mant = little bit [23] @ ieee32FP[9];
```

In this example, the mantissa portion of the *ieee32FP* type is little-endian even though the encapsulating type is big-endian.

3.3. Operators

CISL provides several primitive operators that perform operations on bits and bit arrays. Arithmetic operators return a bit pattern that represents the result of the abstract operation. Boolean operators return a bit representing the truth value of the operation—1 for "true" and 0 for "false". A detailed' account of the operators appears in the CISL manual.

3.4. Classes

CISL also contains facilities for using classes. Classes are used in much the same manner as conventional object oriented languages such as Java, but there are several restrictions on them unique to CISL:

- Classes can contain both data and method-like declarations (see Section 3.4.1)
- There are no interface types, such as in Java,
- The inclusion of a class within another class definition imposes a special restriction. In a general object-oriented language, if the definition of class A includes another class B, the member present once the object of type A is instantiated can have a class of type B or any sub-class of B. However, CISL requires static knowledge of the sizes of every bit array within a type. A situation like that above involving a subclass of unknown size makes this determination difficult if not impossible. Since the containment of one class within another is such a useful construct, the inclusion is allowed with the condition that if a member of one class is another class, the

included member may only be of the included class, not from one if its descendants. The net result of this restriction is that the size and composition of CISL classes are uniquely determined at compile time—there are no references or pointers to other classes. The need to know all sizes at compile time is also the reason why recursive subranges are not allowed.

- Single inheritance is allowed, as are functions as in conventional object-oriented languages.

3.4.1. Attributes

Attributes in CISL are rough analogues to class methods in other object-oriented languages. They are meant to convey properties of a class rather than explicitly executable code. For example, say there exists a class *inst_add* that represents an add instruction in a particular ISA. The machine description we are constructing will be used to generate a simulator and a disassembler. Therefore, we need two attributes to fulfill the needs of these tools: an *effect* attribute that represents the semantics of the add instruction, and an *asm* attribute that represents the textual representation of the instruction. Here is an example of a class with these attributes:

```
class inst_add
{
  big unsigned bit [6] ra;
  big unsigned bit [6] rb;
  big unsigned bit[6] rd;

  effect () {
    addi ()
  }

  asm(ra, rb, rd) {
    "addi"
  }
}
```

The *effect* attribute contains a call to a *mixin*, which is a semantic building block (analogous to a function) described later in this section. Note that the *addi* mixin takes no parameters; the definition is given below. The *asm* attribute returns a format string corresponding to the assembler mnemonic of the instruction. The names and format of these two attributes arise from the manner in which the CoGenT tool set will interpret and use them to produce the simulator and assembler. This is further detailed in Section 4.

3.4.2. Constraints

CISL also allows *constraints* on the value of class members. Constraints represent restrictions on the values of class members. They are useful for representing common semantic actions such as instruction parsing (decoding) and emitting (encoding). For decoding, a constraint validates the identity of the containing class based upon the given constraints on its members (i.e., checking for equality). For emitting, a constraint dictates the values of member fields that must be emitted (i.e., assignment to an output value). This duality of semantics allows the machine description writer to express one of the primary actions of encoding and decoding within a single concise syntax. Note that equality is currently the only constraint that is supported. Here is an example of a class that contains an unsigned bit array, the first two bits of which are each constrained to be 1:

```
class A
{
  big unsigned bit [8] byte;
  big unsigned bit [2] hiBits @ byte [7];
  hiBits = 0x3;
}
```

3.4.3. Mixins

As mentioned previously, *mixins* are semantic building blocks that may be used to implement functionality within attributes. Semantic functionality, i.e., functionality that operates on patterns as opposed to describing them, is very important when attempting to generate tools such as simulators from a machine description. In that case, the semantics of each instruction in a particular ISA would be built up from mixins.

A mixin in CISL is a namespace that contains only methods, which are closer in spirit to methods in the traditional sense (functions). These methods may contain free variables that refer to another class or mixin, with the condition that the CISL compiler must be able to resolve all of the free variables at compile time within the scope of the class that calls the methods of the mixin. At that point, a fully specified method is achieved, and can be used to operate on patterns. Using mixins in this manner is useful for any kind of ISA, but it is especially useful for instruction sets that possess different instructions that utilize common semantic elements such as register access or a specific form of addressing. In our model, a mixin may be set up for each addressing mode and each type of register access, and the machine description writer will not have to specify an implementation of each of these semantic elements in each instruction that needs them. The

writer only has to use the mixin, and CISL will "mix" the mixins correctly to obtain the proper implementation.

Here is a simple example of a mixin named *arith*,

```
mixin arith requires(ra, rb, rd)
{
  addi () { gp[rd] = gp[ra] + gp [rb] }
}
```

This mixin introduces the keyword *requires*, which requires that the three variables *ra*, *rb*, and *rc* be defined in the scope of the class using the mixin. The name of the mixin, *arith*, defines a form of namespace, so that mixins with identical signatures but different method implementations can be used. The body of the mixin defines the method *addi*, which adds the contents of two registers and stores them in a third. The meaning of *gp* is actually part of what is known as the *machine state description*. This will be explained further in Section 4, but suffice it to say that it represents the register file. In Section 3.4.1 concerning attributes, we described a class that uses the method *addi*, but that class was not complete because the source of the method was not specified. Here is the complete definition of that class:

```
class inst_add uses arith
{
  big unsigned bit[6] ra;
  big unsigned bit[6] rb;
  big unsigned bit[6] rd;
  effect () {
    addi ()
    }
 ...
}
```

This class now specifies the source of the *addi* method as the mixin *arith*. The class also defines the three register identifiers required by the mixin, so the method is fully specified at compile time. It is also possible to pass these identifiers into the specification of the mixin in the *inst_add* definition as follows:

```
class inst_add uses arith (ra=rx, rb=ry, rd=rz)
{
  big unsigned bit[6] rx;
  big unsigned bit[6] ry;
  big unsigned bit[6] rz;
  effect () {
    addi ()
    }
 ...
}
```

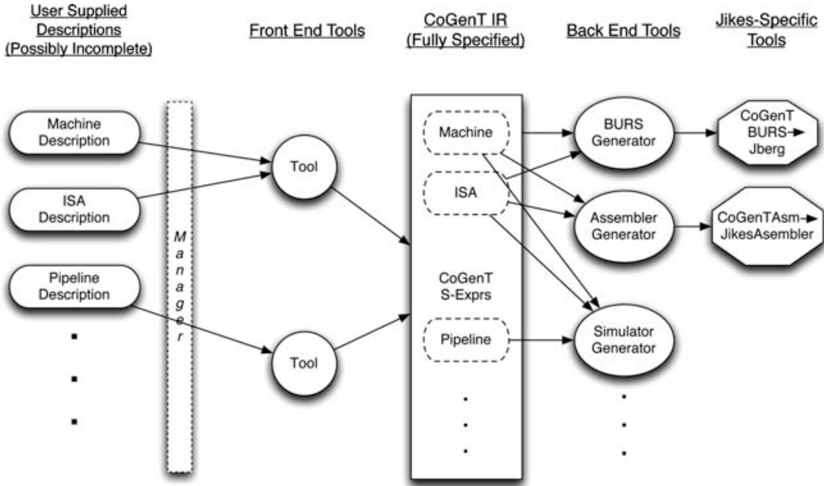

Fig. 1. Architecture of the GoGenT system.

This example simply assigns an alias to variables defined in the class so they may be resolved by the mixin. It should be noted that CISL also has a macro facility that operates like a syntactic mixin. Macros can be applied to classes to create derived classes with new members and constraints. Time and space constraints prohibit a detailed discussion of macros. Interested readers should consult the language manual.[13]

In sum, mixins are semantic building blocks that can be used to build up class attributes. They enable the machine description writer to adjust the granularity of the needed semantic pieces to the most convenient level of detail, and provide an automatic means of integrating them correctly. This way, instead of representing the Cartesian product of the semantic elements, one can simply represent each element and the necessary combinations.

4. CISL AND THE COGENT PROJECT FRAMEWORK

Having laid out in detail CISL we now describe more concretely the system within which this language will exist (illustrated in Fig. 1). The figure shows the flow of CISL descriptions through the CoGenT system. The descriptions of various aspects of a machine (instruction syntax/semantics, pipeline, etc.) written in CISL begin at the far left of the figure. These descriptions will be processed by front-end tools and compiled into an intermediate representation. This intermediate representation

will then be used by back-end tools to generate components to be plugged into the back-ends of compilers and simulators.

Although this paper has focused on the descriptions of instruction set architectures we envision an entire collection of descriptions covering all aspects of a machine, such as architecture pipelines, machine storage locations, calling conventions, and others. These descriptions will have the same flavor as CISL so it is not only easier for the user who must learn the language but also will provide for easier construction of tools that will process those descriptions.

Front-end tools are responsible for parsing the set of machine descriptions as well as performing semantic and type analysis to ensure language correctness. It may be the case that a particular description may reference elements that have been defined in other descriptions. It might also be the case that some description may want to refer to entities defined in several different descriptions depending on the particular machine we are working with. For example, we might want the semantics of instructions defined for architecture A to use the definition of the store for machine A during a particular component generation phase and use a different version of that machine store, say A.2, during another generation phase. To provide for such flexibility in machine description composition a manager, as shown in the figure, resides between the machine descriptions and the front-end tools.

By separating the composition of machine descriptions from the front-end tools we maintain a simple and clean interface between the description languages themselves and the tools that process them. This in turn allows us to create more front-end tools quickly and easily. It is important to recognize in a system such as CoGenT that the ability to mix and match architecture specifications provides for a more flexible environment for exploration and experimentation is just as important as the flexibility of the language in which those specifications are written.

The front-end tools are responsible for resolving all references to variables and types that are defined in the machines descriptions. Once this is accomplished, the front-end tools emit an intermediate representation that is fully resolved according to the input descriptions. This intermediate representation is intended to be in a simple form that allows back-end tools to process it easily and efficiently. Because the complexities of program analysis have been accomplished already by the front-end tools, it allows the back-end tools to focus on the particular components they must generate. This "compiled" form also represents an instance of an architecture produced by the machine descriptions. It may be the case that we wish to produce several different instances that have small variations in order to explore new architectural ideas.

The back-end tools transform the intermediate representation originating from the machine descriptions into components that are to be "plugged in" to existing system tools and frameworks. A sample of an existing compiler framework for which we intend to generate compiler back-end components is the Jikes RVM Java virtual machine. We have considerable experience using and modifying the Jikes RVM, which developed at IBM Research. It is now available as an open source system, with many academic research groups using it for work in compilation. Jikes RVM includes both a baseline compiler (which provides very little optimization) and an optimizing compiler. The baseline compiler compiles Java bytecodes directly into the target machine code, whereas the optimizing compiler has several layers of intermediate representations.

To generate compiler components for the Jikes RVM optimizing compiler we will include a machine description for the IR used in the Jikes RVM as well as for the target architecture. Using the bounded heuristic search technique described by Cattell(3) in the CoGenT back-end tool we will be able to match the semantics of the Jikes RVM IR to the semantics of the target architecture, and to generate BURS rules(14) that map the IR to the target architecture. While the optimizing compiler is designed to be retargetable using BURS rules, the baseline compiler is not designed in a retargetable way. We need to determine how to generate an efficient baseline compiler from semantic descriptions of the Java bytecodes (the "IR" in this case) and the target ISA.

Similarly, we will have back-end tools that will generate components that will plug in to a simulation framework we will develop from scratch. We will base it on existing systems, such as SimpleScalar,(15) but the automatic retargeting that is the essence of our approach means that many pieces will be new. We view a simulator as consisting of many kinds of components. Of these, the ones we propose to generate automatically from ISA and timing descriptions in the back-end tools are instruction semantics and instruction timing. We assume that the remaining semantics and timing components present suitable interfaces to call, and to be called by, the generated instruction-related components. We plan to offer a modest range of options in the simulator framework of semantics and timing components, such as branch predictors, caches, and memory organizations. There will also be a particular emphasis on instrumentation in the simulator frame-work.

5. FUTURE WORK

In the immediate future we will have the CISL implementation complete and well tested. This includes not only the parser but a full set of

front-end tools and a fully specified intermediate representation. In the longer term our intentions are extensions of CISL for describing architectural details, and to design and implement a simulator framework. A major portion of this framework will consist of a collection of retargetable support components (cache models, linkers, loaders, language run-time system, OS interface, etc.). Concurrently, we will be constructing the necessary back-end for consuming our intermediate representation and generating correct and reasonably efficient versions of all target-dependent simulator components. Following the completion of our front-end tools, further CoGenT development will occur on two parallel tracks. One track is concerned with constructing the simulator framework and component generators described previously. The second will progress towards automatic code generator generation from machine descriptions, building on the work of Cattell.[3]

6. CONCLUSIONS

CISL, although inspired by previous efforts, was designed from scratch and is different from other machine description languages in several no-table ways. First, CISL's Java/C like syntax was designed to be sufficiently generic to be useful across multiple sub-domains of the machine-description space. Second, CISL employs a restricted form of object-orientation to aid the description of instructions and machines. Class inheritance provides a convenient way step-wise to define instruction groups (particularly in orthogonal instruction sets). Also, inheritance allows shared features to be defined in parent classes, and thus partial descriptions are easy. Third, CISL is a strongly typed language whose fundamental types are bits and arrays of bits. This single feature alone allows us to describe any binary machine. In addition, the ability of users to define their own types allows the CISL programmer to define complex machine structures both easily and cleanly. Because of these features we believe that CISL is an excellent language for accomplishing the varied and distinct goals of the CoGenT project.

ACKNOWLEDGMENTS

This material is based up on work supported by the Nation Science Foundation under grants ACI-0203895, CCR-0085792, and CCR-0310988. 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.

REFERENCES

1. C. W. Milner, *Pipeline Descriptions for Retargetable Compilers: A Decoupled Approach*, Technical Report CS-99-11, University of Virginia (June 1998).
2. E. C. Schnarr, *Applying Programming Language Implementation Techniques to Processor Simulation*, Ph.D. dissertation., Computer Sciences, University of Wisconsin-Madison (2000).
3. R. G. G. Cattell, Automatic Derivation of Code Generators from Machine Descriptions, *ACM Trans. Program. Lang. syst*. **2**(2):173–190 (April 1980).
4. S. R. of Machine Instructions, Norman Ramsey and Mary F. Fernandez, *ACM Trans. Program. Lang. Syst*. **19**(3):492–524 (May 1997).
5. N. Ramsey and J. W. Davidson, Machine Descriptions to Build Tools for Embedded Systems, *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*, pp. 172–188 (June 1998), available as Springer Verlag LNCS 1474.
6. A. Fauth, J. V. Praet, and M. Freericks, Describing instruction set processors using nML, in *Proc. of the 1995 European conference on Design and Test*, p. 503, IEEE Comput. Soc. (1995).
7. S. Onder and R. Gupta, Automatic Generation of Microarchitecture Simulators, *IEEE Inter. Conference on Computer Languages, ICCL, Chicago, IL* (May 1998).
8. A. Halambi and R Grim, EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability, in *Proc. of the European Conference on Design, Automation and Test*, DATE (March 1999).
9. E. Schnarr, M. D. Hill, and J. R. Larus, Facile: A Language and Compiler for High-Performance Processor Simulators, *ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM* (2001).
10. L. George and A. Leung, MLRISC: A Framework for Retargetable and Optimizing Compiler Back Ends. Availble at http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/.
11. R. Lipsett, C. F. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers (1989).
12. D. Thomas and R Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers (1995).
13. T. Palmer, T. Richards, and E. Walters, CoGenT Langauge Manual, Availble at http://www.ali-cs.umass.edu/cogent.
14. C. W. Fraser, D. R. Hanson, and T. A. Proebsting, Engineering a Simple, Efficient Code Generator Generator, *ACM Lett. Program. Lang. Syst*. **1** (3):213–226 (September 1992).
15. D. C. Burger and T. M. Austin, The SimpleScalar Tool Set, Version 2.0, *Comput. Architecture News*, **25**(3):13–25 (June 1997), extended version available as Univ. Wisc. Comp, Sci. Tech. Rep. 1342 (June 1997).