This is a self-archived, author-created version of this work. The final publication is available at www.springerlink.com: http://dx.doi.org/10.1007/s10766-012-0223-8

International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

ADL-Based Specification of Implementation Styles for Functional Simulators

David A. Penry · **Kurtis D. Cahill**

Received: 20 Mar 2012 / Revised: 8 Aug 2012 / Accepted:

Abstract Functional simulators find widespread use as subsystems within microarchitectural simulators. The speed of a functional simulator is strongly influenced by its implementation style, e.g. interpreted vs. binary-translated simulation. Speed is also strongly influenced by the level of detail of the interface the functional simulator presents to the rest of the timing simulator. This level of detail may change during design space exploration, requiring corresponding changes to the interface and the simulator. However, for many implementation styles, changing the interface is difficult. As a result, architects may choose either implementation styles which are more malleable or interfaces with more detail than is necessary. In either case, simulation speed is traded for simulator design time. Such a tradeoff has become particularly unfortunate as multicore processor designs proliferate and multi-threaded benchmarks must be simulated.

We show that this tradeoff is unnecessary if an orthogonal-specification design principle is practiced: specify *how* a simulator is to be implemented separately from *what* it is implementing and then synthesize a simulator from the combined specifications. We show that the use of an Architectural Description Language (ADL) with constructs for implementation style specification makes it possible to synthesize interfaces with different implementation styles with reasonable effort.

Keywords Simulation · Architectural Description Languages · Domain-specific languages · Model development

D. Penry
Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT 84602, USA
Tel.: 801-422-7665
Fax: 801-422-0201
E-mail: dpenry@ee.byu.edu

K. Cahill
Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT 84602, USA
E-mail: mathematica_k@hotmail.com

1 Introduction

Microprocessor and SoC architects use simulators to evaluate new ideas, explore the design space, and validate the behavior of new microprocessors and embedded systems. One important class of simulators is *functional simulators* – simulators which model the instruction-set behavior of a microprocessor. These simulators are useful for early software development, but also find widespread use as subsystems within microarchitectural simulators which provide near-cycle-accurate predictions of performance.

Ideally, architects would like to explore substantial portions of the design space. This can only be possible if development time for simulators is short and simulation speed is high enough to allow simulation of large samples of code. The use of large samples of code and multiple simulation runs of that code is becoming even more necessary as multicore processors proliferate; multi-threaded workloads need to be simulated multiple times for many cycles in order to address variability in the simulations [2]. Functional simulation speed can easily become a bottleneck, particularly when the microarchitectural performance prediction model is simple or uses sampling techniques [29].[1]

The simulation speed of a functional simulator is closely tied to its *implementation style*. We define implementation style as "the means by which the simulator emulates target instructions." Researchers have introduced many implementation styles; familiar examples include interpretive simulation, compiled simulation, and binary translation. Architects must choose which implementation style to apply in their simulator, making tradeoffs between simulator development time, simulation speed, and simulator capabilities.

When a functional simulator is combined with a timing simulator to form a microarchitectural simulator, the simulation speed of the functional simulator is also closely tied to the interface which it presents to the timing simulator. The interface defines the means of control that the timing and functional simulator offer each other and the information which is communicated between them. This interface is affected by the needs of the timing simulator and the overall organization of the microarchitectural simulator. Some microarchitectural simulator organizations will require more or less detailed information and more or less detailed control than others. For example, one timing simulator may wish to control the time at which operands are read and written, but another may not need this control. The level of detail may evolve over time as the microarchitectural design space is explored. For example, early in the design process, timing simulators may abstract many details of the timing and require little information from the functional simulator. As the design becomes better-specified, the timing simulators become more detailed and may then require more information. Furthermore, a single timing simulator may require multiple levels of information or control – e.g., one for detailed simulation and one for fast-forwarding.

The requirement to provide multiple, potentially evolving interfaces increases the difficulty with which an implementation style can be applied. In higher-performing

[1] When sampling is used, most of the microarchitectural simulator's execution time may be spent in fast-forward mode between samples and functional simulation speed may be more influential than timing simulation speed.

implementation styles such as binary translation, the code which must be generated will be different for each interface, leading to much more implementation complexity. As an example, [26] reports the difficulties involved in changing QEMU [6], a binary-translating simulator, to support speculation and provide more instruction information in its interface.

This complexity may lead simulator developers to either choose easier – and often slower – implementation styles or to avoid interface changes which would otherwise be beneficial. In either case, simulation speed is traded for development time.

The goal of this work is to obviate the choice between functional simulator speed and development time when an implementation style must support multiple interfaces. We do so by practicing a fundamental design principle:

Orthogonal-Specification Principle
Specify how *a simulator is to be implemented separately from* what *it is implementing and synthesize a simulator from the combination of the specifications.*

1.1 Contributions

The primary contributions of this work are:

- The Orthogonal-Specification Principle for functional simulators.
- A description of how Architectural Description Languages (ADLs) – domain-specific languages which provide architects with constructs to specify instruction behavior – can be extended to enable orthogonal specification of implementation styles. No previous ADL has provided a set of constructs which support orthogonal specification.
- A case study using such an ADL demonstrating the potential of orthogonal specification.

By adopting the Orthogonal-Specification Principle and ADLs which are extended to support it, architects will be able to easily specify implementation styles and apply them to create simulators with multiple, evolving interfaces – thus improving both simulator speed and development time and allowing exploration of a greater portion of the design space with longer multi-threaded benchmarks, leading to improved designs.

1.2 Organization of this article

This article is an extended version of [22] which adds a detailed example of an implementation style description with commentary as well as an appendix describing the ADL used in this work. The example implementation caches decode information and support multiple simulated threads of execution.

Section 2 provides background explaining implementation styles, functional simulator interfaces, and ADLs. Section 3 introduces the Orthogonal-Specification Principle. Section 4 describes ADL support for this principle. Section 5 evaluates the effectiveness of this principle. Section 6 gives an extended example of the specification

of an implementation style. Section 7 concludes. Appendix A provides a description of the ADL used in this article.

2 Background

2.1 Implementation Styles

Researchers have proposed many implementation styles for functional simulators. The most notable are listed below.

Interpretive simulation is the simplest of the implementation styles. The simulator fetches, decodes, and executes each dynamic instruction. The heart of the execution code is usually a switch statement based on the result of decoding or an indirect call to an instruction-specific execution function (called an *instruction function*). For example, an ADD instruction has a corresponding ADD function which performs its behavior. This style is able to easily support self-modifying code. However, these simulators tend to be slow.
Static compiled simulation [18,30] translates each static instruction in a target program into high-level (e.g., C) or assembly-level code through a pre-processing step and compiles said code to form the simulator. An ADL compiler would actually synthesize the compiler which translates target programs into simulators, not the simulator itself. Simulators implemented using this style can be very fast, as all fetch and decode behavior is performed at simulator compile time, instruction fields can be taken as constants, and inter-instruction optimizations are often possible. However, such simulators are unable to support self-modifying code, and large target programs can cause host compilers to take excessive time or memory if care is not taken when generating the code [3,4,9].
Dynamic compiled simulation [12,23] performs the same kind of translation as static compiled simulation, but at run time instead of compile time. The simulator generates code in a high-level language, asks an external compiler to create a shared object from that code, and then loads the shared object. Self-modifying code is supported by invalidating the compiled code when an instruction changes. The overhead of calling an external compiler is quite high, though the resulting code is fast for the same reasons it is fast in static compiled simulation.
Binary-translated simulation [10,17,19,28] is similar to dynamic compiled simulation, however a dynamic code generator in the simulator is used instead of an external compiler. In general, binary translation is seen as a very high-performance method of simulation with the same optimization possibilities that exist in static compiled simulation. There is usually a tradeoff to be made between the quality of the generated code and the speed of code generation. Many high-speed binary translation schemes attempt to only translate code which is sufficiently "hot" in order to avoid code generation costs for infrequently executed code.
Just-in-Time cache compiled (JIT-CCS) simulation [5,20] adds caching to interpretive simulation. As a static instruction executes for the first time, the simulator caches the results of decoding – the instruction function and pointers to the operands. The simulator reuses this decoding when it executes the instruction in the future. Thus

Table 1: Microarchitectural simulator organizations and required levels of detail

Organization	Informational detail	Semantic detail
Functional-First	**Moderate** register numbers, branch resolution, effective addresses	**Low** single call per instruction or basic block
Timing-Directed	**High** register numbers, branch resolution, effective addresses, operand values	**High** individual operand read/write, steps of execution
Timing-First	**varies**	**Low**: single call per instruction
Speculative Functional-First	**Moderate** register numbers, branch resolution, effective addresses	**Low** single call per instruction or basic block

decode happens only once per static instruction. Self-modifying code is supported by invalidating the cache when an instruction changes.

Instruction-set compiled (ISCS) simulation [25] uses a pre-processor which initializes the state of a JIT-CCS-like cache for a target binary. This pre-processor also specializes the instruction functions for the instructions actually present in the program by selecting some bits from the instruction encoding and making those bits a constant in the specialized instruction. For example, the ARM instruction set benefits from specialized instructions whose predicate is always true. An ADL compiler would synthesize the pre-processor, which is then run to produce simulators. Self-modifying code is supported by invalidating the cache when an instruction changes and falling back to interpretive simulation for that instruction. Large target programs can cause host compilers to take excessive time or memory.

Hybrid compiled simulation [24] reduces the compile-time overhead of ISCS simulation by removing the initialization of the decoded-instruction cache. The pre-processor generates the same kinds of specialized instruction functions, but also generates a specialized decoder which can select the specialized execution functions when appropriate and is used at runtime to fill the cache. An ADL compiler would synthesize the pre-processor used to create the decoder and specialized instruction functions. Self-modifying code is supported as in ISCS, but the specialized decoder and the specialized instruction functions are used as the fall-back mechanism.

2.2 Functional simulator interfaces

Functional simulators are often paired with timing simulators to create microarchitectural simulators. In such simulators, the timing simulator makes calls to the functional simulator through some interface; these calls perform instruction semantics and return information about instruction execution. We refer to the amount of information present in an interface as its *informational detail*. We refer to the amount of control present in the interface as its *semantic detail*. Differing levels of detail may affect the number of functions, function signatures, and data structures comprising the interface.

The design of the interface and its level of informational and semantic detail are affected by the needs of the timing simulator and the overall organization of the mi-

croarchitectural simulator. A taxonomy of microarchitectural simulator organizations was first introduced in [16] and has since been extended [8]. Table 1 lists the various organizations as well as the typical levels of informational and semantic detail required to support each organization. Multiple interfaces may be required; e.g., microarchitectural simulators which use sampling may require a "fast-forwarding" interface which simply executes N instructions and reports no information; this interface is provided in addition to the "detailed" interface.

2.3 ADLs

Architecture Description Languages (ADLs) are commonly used to provide descriptions of an instruction set architecture which can be synthesized into a functional simulator or used to retarget a compiler. Most ADL compilers synthesize simulators using a single implementation style; interpretive simulation is the most common style. A few ADL compilers have been extended to synthesize multiple styles. Many ADLs have previously been proposed; we mention only those which have some relevance to implementation styles.

D'Errico and Qin [11] describe a simple ADL which allows synthesis of both interpretive and dynamic compiled simulators from the same language description. Specific language constructs for describing styles are not provided, though they describe some instruction constructs which make it easier to perform the synthesis of both styles.

Leupers, Elste, and Landwehr [15] describe the generation of both interpretive and static compiled simulators from a single description; however in this case the description is of a microarchitecture and the instruction set is inferred from the description.

3 The Orthogonal-Specification Principle

Developing a functional simulator is conceptually straight-forward – simply implement the ISA manual. However, implementing a simulator with multiple interfaces can be rather complex. Each interface requires its own implementation, and the implementation must take into account details of the interface such as data structure definitions and function signatures. The most obvious way to do this is to implement different functional simulators for each interface. Obviously such an approach takes significant time, leads to large amounts of code duplication, and requires extensive validation.

The creation of multiple interfaces can be greatly simplified by specifying a single highly-detailed interface *once* and then synthesizing lower-detailed interfaces from that specification as needed. This concept was introduced in [21] as the Single-Specification Principle and was shown to lead to very short development times – mere minutes to specify a new interface – as well as easier validation and improved simulation speed.

An underlying assumption of [21] was that an implementation of any derived lower-detail interface could be provided without significant user intervention. For

implementation styles which have been built into a tool chain, this is clearly possible. However, such tool chains limit architects to using only the styles built into the tool chain; they cannot use newly developed implementation styles and cannot tune an implementation style for their needs.

A better approach is to allow architects to describe styles of implementation. We call such an approach the *orthogonal specification* approach and can state it as an additional design principle to be practiced in conjunction with the Single-Specification Principle:

Orthogonal-Specification Principle
Specify how *a simulator is to be implemented separately from* what *it is implementing and synthesize a simulator from the combination of the specifications.*

The simulator design complexity problems which multiple interfaces and multiple implementation styles present are solved by the joint practice of the Orthogonal-Specification and Single-Specification Principles. The architect only specifies a particular implementation style once and then is able to implement multiple interfaces with that style. The architect does not need to change instruction specifications or interface specifications to support different implementation styles.

3.1 Applying the Orthogonal-Specification Principle

Orthogonal specification can be seen as a form of aspect-oriented programming [13]; implementation styles and interfaces are aspects of the system while instructions are the objects. Aspect-oriented programming is made much easier with tool and/or language support. Therefore, we propose that constructs for specifying and synthesizing implementation styles be added to ADLs. The resulting methodology for simulator design, extended from that of [21], looks like:

1. **Specify the instruction set at a high level of detail.**
2. **Describe a timing simulator interface.** This interface is used for initial debugging of the instruction set description. We recommend an interface with a low level of semantic detail and a high level of informational detail. An interface with a low level of semantic detail such as one function call per instruction has proven easy to debug because a single breakpoint can be used to stop at a particular instruction and because all of the semantics of the instruction are in one place in the generated simulator code. Use of an interface with a high level of informational detail eases debugging by directly providing more information; indeed, we have found that much debugging can be done by simply printing all operand values for each instruction.
3. **Describe the interpretive implementation style.** This implementation style is the easiest to understand and debug. Our tool chain includes interpretive simulation as a built-in style so the user need not specify it.
4. **Synthesize and validate the resulting simulator.** The validation should be done by running a large number of programs whose output can be tested; ideally an ISA validation suite would be used.

5. **Describe and validate additional interfaces and/or implementation styles.** Interfaces are validated using previously validated implementation styles while styles are validated using previously validated interfaces. These validations need not be as extensive as the original validation; the instruction semantics are already known to be good and the only mistakes which can be made are in interface or implementation style specifications.
6. **Repeat step 5 as necessary.** We emphasize that it is neither necessary nor desirable to specify or validate all interfaces or implementation styles *a priori* when designing a microarchitectural model.

3.2 ADL support for orthogonal specification

To support such a design flow, an ADL must have constructs which allow the specification of implementation styles. These specifications must be able to:

1. specify the code to be generated when the simulator is synthesized.
2. query the interface and instruction definitions.
3. extend the interface and instruction definitions.
4. support the per-static-instruction code specialization used in many implementation styles.

We now discuss each ADL requirement in turn.

3.2.1 Code generation specifications

An implementation style specification must be able to tell the ADL compiler *how* to generate the code of the simulator (or for styles such as static compiled simulation, the simulator compiler) from the specifications of instruction decoding and behavior. This code may include datatype definitions, support functions for the style, instruction functions which perform semantics, and functions which themselves may generate code. Some simulator code will need to be generated only once, but other simulator code may need to be generated once per interface.

The structure of the code to be generated depends upon the implementation style, and thus the specification must be able to provide this structure. For example, the specification of an interpreted implementation style need merely indicate that the appropriate instruction function be called after fetch and decoding. On the other hand, the specification of a binary-translated implementation style needs to enable an ADL compiler to generate code which looks up the instruction's host translation in a code cache. If it is not found, then the generated code must fetch and decode the instruction, select its semantics, send those semantics to a dynamic code generator, and cache the results.

3.2.2 Querying interfaces and instructions

Orthogonal specification requires that the implementation style be independent of the interfaces. In other words, the implementation style specification may not make assumptions about the signatures and datatypes of interfaces. However, it does need to

know about them in order to implement them. Similarly, the implementation style specification should not make assumptions about the semantics of individual instructions, but does need to know what they are.

The instruction semantics and interface signatures may need to be inserted directly into the generated code or their text may need to be made available to the generated code as strings (e.g. for static or dynamic compiled simulation).

3.2.3 Extending interfaces and instructions

Some implementation styles need to perform calculations based upon the results of decoding. For example, the ISCS and hybrid compiled simulators need to calculate which bits of the instruction encoding should be specialized. While it would be possible to write a function to perform this decoding as part of the implementation style description, it is much more convenient to define this additional per-instruction calculation as auxiliary instruction semantics. These semantics, as they are not part of normal instruction execution, should be accessed through an additional interface. Thus both the interfaces and the instruction semantics need to be extensible within an implementation style description.

3.2.4 Specialization support

Implementation styles such as static compilation, ISCS, and binary translation generate specialized code for each static instruction. This specialization uses the values of particular bits of the instruction encodings (e.g. the condition predicate in the ARM instruction set) and/or the instruction address. The implementation style description needs to be able to generate code with specializations in place.

4 Supporting the Orthogonal-Specification Principle

We now illustrate how an ADL might meet the requirements of Section 3.2 using an ADL named LIS which we have extended to support orthogonal specification.[2] The simulators which are generated from this LIS description are specialized for use with the Liberty Simulation Environment (LSE) [27]. As the implementation details of LIS are neither the focus nor a contribution of this paper, explanations of all LIS constructs are not given here but can be found in Appendix A. We wish to emphasize that *equivalent constructs could and should be added to other ADLs* and that neither the syntax nor the semantics of these constructs should be construed to form an "ideal" set of constructs; indeed, the ideal set of constructs for a particular ADL would be that set which best accords with the feel and philosophy of that ADL.

As we are now about to introduce specific LIS constructs, it becomes necessary to distinguish between general concepts such as interfaces and implementation styles, the LIS keywords which support their specification, and particular specifications using the LIS keywords. General concepts will be shown in normal text; new concepts

[2] LIS was developed specifically to implement the Single-Specification Principle and is partially described in [21].

will be italicized on their first occurrence. LIS keywords are given in **small boldface type** and introduce statements with the same name. Particular uses of the keywords within descriptions will also be referred to as statements, but should be clear from context.

LIS identifiers will appear in *small slanted type*, and C++ keywords and identifiers in small type. However, many LIS identifiers are also C++ identifiers in generated code and will appear within C++ code inside of example LIS descriptions in *small slanted type*.

4.1 Code generation specifications

The LIS compiler takes a LIS specification as input and outputs several C++ files – a public header file used by clients of the functional simulator, a private header file, a support code file, a makefile, and source files for each implementation style. These files are generated from templates; the user can view the process of writing a LIS description as a process of filling in sections of the template. Figure 1 provides an outline of the templates for each generated file. Portions of the outline in upright type are generated by the LIS compiler. Portions of the outline in slanted type are supplied by the user and are known as *codesections*. Each codesection has a name indicated in slanted type; their intended uses are given as a comment following the name.

LIS provides two constructs for filling codesections. These constructs are not found in other ADLs. The **codesection** statement is used for code which is to be inserted *once* into the generated simulator. The **generator** statement is used for code which is to be inserted once per interface function implemented using an implementation style. The user may define additional, non-standard codesections and indicate where they are to be inserted.

The following LIS code illustrates these concepts. It comes from the ISCS specification and defines a data structure which is used to cache pointers to the instruction functions for each interface function. The **generator** statement on lines 1-3 inserts its body into the user-defined *iscs_structdef* codesection once for each interface function. The **codesection** statement on lines 5-9 inserts its body, which includes the *iscs_structdef* codesection, once into the private header file. The odd-looking macros prefixed by %% will be explained later.

```
1  generator iscs_structdef {
2    void (*%%NAME())(REGPTRS_ptrsP_t, %%PARMS());
3  }
4
5  codesection private {
6    struct LIS_etableC_t {
7      LIS_CODESECTION(iscs_structdef)
8    };
9  }
```

Note that not all of the code of the simulator needs to be in the LIS specification; the specification only requires the code which could be influenced by the interface or instruction set. The generated code can and usually does call support libraries. Such libraries increase code reuse and can greatly reduce simulator design time. For example, the description of the binary translated style that we have defined relies upon LLVM libraries [14] to perform code generation.

headers	// headers to include
LIS-generated public options	
Decoding result types	
Opcode names	
earlypublic	// types
LIS-defined types	
public	// types and inline functions

(a) Public header template

privateheaders	// headers to include
declarations of decode tables	
private	// types, variable declarations
inline functions for accessing state	
per-interface options	

(b) Private header template

support	// Support functions and variables
Decode tables	

(c) Support code template

LIS_SRCFILES=list of files	
makefile	# Rules needed to make additional files

(d) Makefile template

stylename_*headers*	// headers to include
stylename_*prologue*	// style-specific functions and variables
Instruction functions	
Decoding functions	
Intermediate interface functions	
stylename_*epilogue*	// interface functions

(e) Style-specific code template

Fig. 1: Templates for generated C++ files

Implementation styles are named within a LIS specification by grouping **codesection** and **generator** statements within named **style** statements.

4.2 Querying interfaces and instructions

LIS provides knowledge of interfaces and instructions to the implementation style through macros which are placed within codesections. These macros expand into function signatures, function names, data type definitions, and instruction semantics; they provide the link between the very general code placement constructs of the previous subsection and the specific details of implementing a simulator. The macros and their functionality are listed in Table 2. Detailed explanations are not given, as they are highly specific to the form in which LIS specifies instructions and interfaces; the purpose of the table is to give an idea of the kinds of macros that are needed. While we feel and demonstrate that this list of macros provides the information needed to support a very wide variety of implementation styles, from binary translation to interpreted to static-compiled, it is possible that additional macros will need to be added as new styles are invented.

The following example illustrates how these macros are used; this example comes from the interpretive simulation specification and defines how to generate an interface function. The *interp_epilogue* codesection is placed at the end of the generated source file for the interpretive style. A **generator** statement is used so that an interface function is generated for each function in each interface implemented using the interpretive style. Lines 2-6 define the form of interface functions. The **%%NAME()** and **%%PARMS()** macros provide the correct function name and signature. The **%%DECLS()** macro on line 3 defines all intermediate and operand value variables that the instruction semantics within this function may need. The **%%BEFORE()** macro (line 4) expands to the semantics (code) shared by all instructions, e.g. fetch and decode. The **%%AFTERPTR()** macro (line 5) expands to a lookup of a table holding pointers to instruction functions. The lookup is indexed by the results of decoding. The **%%ARGS()** macro provides the correct arguments to the instruction function.

```
1  generator interp_epilogue {
2    void %%NAME() (%%PARMS()) {
3      %%DECLS()
4      %%BEFORE()
5      %%AFTERPTR() (%%ARGS());
6    }
7  }
```

Table 2: LIS code generation macros

Name	Purpose
AFTER()	Post-decoding instruction semantics
AFTERARGS()	The arguments to an instruction function
AFTERPARMS()	The parameters of an instruction function
AFTERPTR()	Looks up an instruction function
AFTERREC()	Looks up all instruction functions for an instruction
ARGS()	The arguments to an interface function
BEFORE()	Semantics which come before decoding
BUILDSET()	The name of an interface
COMMA()	Inserts commas in parameter lists if needed
DECLS()	Declares instruction fields and operands
ENTRYTEXT()	Looks up the text of an instruction function
ENTRY()	A name of an internal function used to implement an interface function
IFAFTER()	Inserts its arguments only if the interface function contains post-decoding semantics
IFNAFTER()	Inserts its arguments only if the interface function does not contain post-decoding semantics
NAME()	The name of an interface function
PARMS()	The parameters of an interface function
RTYPE()	The return type of an interface function
STYLE()	The name of the implementation style
TEXT()	Quoted text of its argument
TOKEN()	A variable holding the results of decoding
TOKENSWITCH()	A switch statement with cases corresponding to instructions and their semantics

4.3 Extending interfaces and instructions

LIS uses *open* constructs – constructs which can be modified after their initial declaration. Thus the implementation style can add semantics to entire classes of instructions without requiring changes to the original instruction specification. The following example from the PowerPC ISCS implementation adds a particular bit of a branch instruction's encoding to the specialized bit computation:[3]

```
1  action b +decloc = { specbits |= AA << 1; }
```

Open constructs also allow LIS implementation styles to add intermediate values which can be used for instruction execution in conjunction with new semantics. The following example from the PowerPC JIT-CCS specification declares a new intermediate value regptrsP which holds cached pointers to operands. It then overrides the semantics of fetching the first operand of an instruction from the register file; instead of indexing into the register file using the operand specifier in the instruction, the cached pointer is dereferenced.

```
1  field regptrsP CACHE_REGPTRS_ptrsP_t;
2
3  action r2_nOE_rC_1 @ fetchOp1Step = {
4    src1 = *(regR_t *)regptrsP->regPtr1;
5  }
```

Note that the constructs which were used here are not new; their equivalents are present in many ADLs. The point we wish to emphasize is that the constructs need to be open, as closed constructs would require users to modify the original instruction specification.

4.4 Specialization support

It is convenient to generate the specialized functions in a "nested" form which calls normal instruction functions. The specialized function first sets the values to be specialized to constants. It then calls the normal instruction function, which has its parameter list extended to include the specialized values. At compile time, the call is then inlined and the constant specialized values propagated.

LIS provides three new constructs which support specialization. The **entry** statement allows the signature of instruction functions to include parameters which are not part of the signature of the interface function. It also allows the instruction functions to have a different return type from that of the corresponding interface function. The **specialize** statement declares which intermediate values, operands, or fields of the instruction encoding have been specialized. These specialized values are then added as parameters to the instruction functions. Typically the specialized function sets the values of these fields to constants. The **exclude** statement is a simple declaration which

[3] The **action** statement is used to assign semantics to an instruction or class of instructions (Appendix A.3.4). The first argument is the instruction name or class name while the second argument is the label of the *step*; interface definitions may specify which steps to include in interface functions.

removes semantics from the **%%BEFORE** and **%%AFTER** macros. This statement is useful for removing semantics such as decoding which are implied by the specialization and do not need to be done at run time.

The following example from the PowerPC ISCS implementation style shows how these constructs work together. On lines 1-4, the **entry** statement adds a parameter to all instruction functions; this parameter points to the cached operand pointers. It also defines the form of the instruction functions. Lines 6-7 define the instruction encoding fields which are to be specialized. Line 9 excludes all the fetch and decode semantics from the instruction functions. Finally, lines 11-27 are the code which actually generates specialized functions. The LIS compiler places this code into the generated pre-processor function which processes instructions. That function, at pre-processing time, outputs a function which assigns constant values to a number of instruction encoding fields and then performs the instruction's semantics.

```
1   entry (void) (CACHE_REGPTRS_ptrsP_t regptrsP) {
2     %%BEFORE();
3     %%AFTERPTR()(%%AFTERARGS());
4   }
5
6   specialize AA, BI, BO, CRM, d, FM, IMM, MB, ME, NB, OE,
7              Rc, Rc2, SH, SIMM, UIMM, L, DS;
8
9   exclude 0:findOpcodeStep;
10
11  generator ISCS_generators {
12    cstream
13      << "void LIS_" << templateno << '_'
14      << %%TEXT(%%NAME())
15      << "(CACHE_REGPTRS_ptrsP_t regptrsP, "
16      << %%TEXT(%%PARMS())
17      << ") {\n"
18      << %%TEXT(%%DECLS())
19      << "\tconst unsigned sbits = 0x" << std::hex
20      << specbits << std::dec << "U;\n"
21      << "\t const unsigned AA = (sbits >> 1) & 0x1;\n"
22      << "\t const unsigned BI = (sbits >> 16) & 0x1f;\n"
23      << "\t const unsigned BO = (sbits >> 21) & 0x1f;\n"
24       ... // other specialized fields
25      << %%ENTRYTEXT(ii.decodetoken) << "\n"
26      << "}\n";
27  }
```

5 Evaluation

We evaluate the effectiveness of the Orthogonal-Specification Principle by specifying simulators with three interfaces with different levels of detail and a variety of implementation styles.

5.1 Instruction sets

We used LIS to describe two instruction sets: user-mode ARM v5, and user-mode 64-bit PowerPC. The ARM instruction set description is contained in 2272 lines of LIS code, while the PowerPC description uses 3987 lines of LIS code. Operating system

Table 3: Size and development time of style and interface descriptions

	Lines of LIS code[a]		Development time (days)	
	ARM	PowerPC	ARM	PowerPC
Styles				
Interpretive	20	19	< 1	< 1
Static compiled	233	227	5 *	1
Binary translation	248	249	2	21 *
JIT-CCS	415	443	8 *	2
ISCS	736	772	4 *	1
Hybrid compiled	530	556	3 *	1
New: Binary translation + JIT-CCS	618	686	1 *	1
New: Binary translation + JIT-CCS + ISCS	742	778	2 *	1
Interfaces				
Minimal	8	8	< 1	< 1
Decode	10	10	< 1	< 1
Steps	21	23	< 1	< 1

[a] Excludes comments and blank lines.

emulation and emulated memory are handled by C++ libraries called from the synthesized simulator. The ARM description required approximately 40 hours to develop and debug, while the PowerPC description required approximately 60 hours. Neither instruction set description required subsequent modifications beyond the addition of a handful of LIS_CODESECTION hooks to support the implementation styles.

5.2 Implementation styles

For each instruction set, we used LIS to describe six implementation styles from the literature: interpretive, static compiled, binary translation based upon the LLVM compiler framework [14], JIT-CCS, ISCS, and hybrid compiled. Table 3 shows the size of each style description as well as the time approximate required to create and debug it. The first instruction set to be implemented using each style is marked with an asterisk. In all cases the style specification is fairly small; note that the ISCS and hybrid compiled styles share LIS code for manipulating operand pointers with JIT-CCS and this shared code (356 lines and 378 lines for ARM and PowerPC, respectively) is counted multiple times. Additional library code not written in LIS supports binary translation through LLVM.

The development time for a single implementation style ranged from less than an hour (interpretive) to 21 work days (binary translation). The difference in development times stemmed primarily from the relative complexities of the implementation styles; figuring out how to harness LLVM was much more difficult than creating an interpreter. Note that the reported times are only approximate; any work taking place on a particular style on a particular day was counted as one day, though the work may not have required a full day. Indeed, several of the style descriptions reported as requiring one day were in fact developed on the same day. LIS development occurred concurrently with the style development for binary translation and the binary translation development time for PowerPC includes the development of a common

simulation infrastructure component to be shared among LLVM-based binary translators.

One clear trend the development times demonstrate is that porting a style from one ISA to another requires significantly less work than the initial development of the style. We also found reuse of LIS code between implementation styles to be easy; the shared operand pointer manipulation code previously mentioned is a good example.

The use of an ADL with implementation style support also eases new style creation. As an example, we created two additional implementation styles which combine binary translation with JIT-CCS or ISCS. Designing and debugging these new styles required no more than two days. The size and development time of these specifications are also shown in Table 3.

The first new implementation style caches decode information for cold code until it is determined to be hot enough for binary translation. The LIS description is a straight-forward merging of the descriptions of binary translation with JIT-CCS: decode information is added to the code cache and the template for generated code adds specialization on the operand pointers.

The second new implementation style is prompted by the additional observation that the overhead of optimizing and translating code is very high in LLVM; if the number of translations could be reduced, the simulator could be faster. The ISCS style reduces static compilation time by limiting the amount of code created to just specializations of the instruction execution functions. We merge this idea with binary translation in the second variant to only translate specializations which are then reused. We also use the JIT-CCS technique for not-yet-translated code. This LIS description is again a straight-forward merging of the first style's description with the ISCS compilation style's description.

5.3 Interfaces

We used LIS to describe three interfaces for each simulator. The first interface (**Minimal**) is called once per dynamic instruction and provides only minimal information: whether a fault occurred and the next PC. This interface is suitable for fast-forwarding. The second interface (**Decode**) is called once per dynamic instruction and provides the minimal information plus effective addresses, branch direction and targets, instruction classification, and operand decoding information. It is also speculative: the results of instructions can be rolled back. This interface is appropriate for a speculative functional-first simulator architecture [7]. The final interface (**Steps**) is called seven times per dynamic instruction – once for each of a set of major steps of execution – and reports all the information of the **Decode** interface plus operand values. This interface would be appropriate for a timing-directed simulator architecture [16]. The size of these descriptions is also shown in Table 3. We required only a few minutes to create these interfaces and did not need to modify them to support different implementation styles. Neither did we need to modify the implementation styles to support different interfaces.

Table 4: Average ARM simulation speed in MIPS – standard deviation shown in parenthesis [22]

Style	Minimal	Decode	Steps
Interpretive	18.03 (1.75)	12.81 (0.99)	9.34 (0.77)
Static compiled	34.79 (13.38)	10.39 (4.51)	5.46 (1.62)
Binary translation	30.18 (7.13)	19.59 (5.49)	5.03 (0.72)
JIT-CCS	29.10 (4.59)	21.47 (3.07)	10.73 (1.03)
ISCS	35.97 (9.03)	17.99 (4.12)	7.29 (1.42)
Hybrid compiled	35.73 (5.83)	25.79 (4.47)	11.08 (1.19)
New: Binary translation + JIT-CCS	30.28 (7.72)	19.91 (5.96)	4.91 (0.73)
New: Binary translation + JIT-CCS + ISCS	30.04 (6.33)	18.30 (4.65)	5.42 (0.55)

Table 5: Average PowerPC simulation speed in MIPS – standard deviation shown in parenthesis [22]

Style	Minimal	Decode	Steps
Interpretive	19.82 (1.52)	13.06 (0.58)	10.92 (0.49)
Static compiled	31.18 (12.65)	12.28 (4.38)	6.18 (1.84)
Binary translation	30.57 (6.59)	18.60 (4.48)	5.60 (0.59)
JIT-CCS	34.42 (3.9)	28.12 (2.83)	12.62 (0.49)
ISCS	37.91 (8.78)	18.39 (3.44)	8.88 (1.39)
Hybrid compiled	36.19 (4.59)	30.8 (4.16)	12.06 (0.55)
New: Binary translation + JIT-CCS	30.73 (6.27)	19.49 (4.58)	5.59 (0.68)
New: Binary translation + JIT-CCS + ISCS	33.41 (5.05)	20.69 (3.71)	6.30 (0.49)

5.4 Results

We measured the speed of each combination of interface and implementation style by running the SPEC CPU2000int benchmarks using reference inputs. The simulators were run on systems equipped with two 2.8 GHz Intel Xeon X5560 processors and 24 GB of memory and were compiled using gcc 4.1.2 with flags `-g -O2`. For the static compiled, ISCS, and hybrid compiled styles, a distinct simulator was created for each benchmark. Tables 4 and 5 report the average speed of each combination measured in millions of simulated instructions per host second. They also show the standard deviation of the speeds.

For the **Minimal** interface, we see overall results in general agreement with previous work: all styles are much better than interpretive and ISCS outperforms hybrid compiled which outperforms JIT-CCS. The speed of the static compilation and binary translation styles is much lower than that reported previously, achieving speeds only comparable to the other implementation styles; this difference in speed stems from differences in the level of detail of the interface. In previous work, static compilation and binary translation are rarely run on an instruction-by-instruction basis; instead, one call to the simulator simulates an entire basic block, trace, or program and reports no information about the instructions. For comparison, we have generated binary translating simulators that execute one basic block per call and do not report any information[4]; the resulting average simulation speeds are 77.1 MIPS for ARM and 70.58 MIPS for PowerPC. These speeds are comparable to those reported in [26] for binary translation, but are still lower than some previously reported speeds because

[4] These simulators required writing both a new implementation style description and a new interface description capable of handling multiple instructions per call.

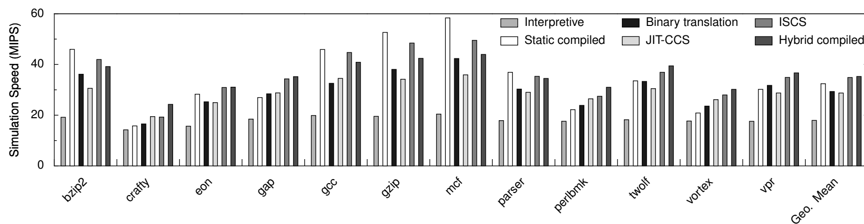

Fig. 2: Simulation speed of ARM minimal interface [22]

we do not use certain optimizations such as branching directly between translated functions.[5] The LLVM binary translator also has very high overhead and the threshold for considering code to be hot and thus worthy of translation has not been tuned. We do not believe that code quality is an issue; manual inspection of the statically compiled simulator binaries and dynamically generated code indicates that the code quality is high in both cases: instruction functions simply load their operands, perform their operation, and then store the results, with any specialization showing up as constants in the binary code and with dead code paths (such as checks for faults on non-faulting instructions) optimized away.

Figure 2 shows the individual benchmark results for the ARM simulators with the **Minimal** interface. There are rather large differences among the benchmarks; not only do the speeds vary, but the ordering of the styles changes.

The **Decode** interface provides enough detail to be useful in a microarchitectural simulator. However, this additional detail comes at a price. First, all speeds are lower than **Minimal** due to the additional work which must be done to report decoding information. Second, the benefit of non-interpretive styles decreases because the additional work is not accelerated as much by the more sophisticated styles. Static compiled simulation is now slower than interpretive simulation and ISCS, which also has a large static compilation component, is worse than JIT-CCS. This difference likely stems from an increase in code size as more information is required.

The **Steps** interface provides a high level of detail and control to a microarchitectural simulator. For this interface, static compilation, binary translation, and ISCS perform exceedingly poorly – worse than interpretive. This poor performance is primarily due to the simulators having a much larger instruction cache working set. This bloated working set stems from more work to report operand values, lost optimization opportunities, and overhead in function prologues and epilogues.

The new implementation styles which combine binary translation with other implementation styles show mixed results. They usually improve slightly upon binary translation, especially as the level of detail increases, but are occasionally worse. Neither presents a compelling enough argument to suggest that they be adopted. We wish to emphasize that the use of an ADL supporting the Orthogonal-Specification Principle enabled us to quickly determine that these styles are not interesting after only a small amount of work.

[5] Note also that such optimizations would not be possible at high levels of semantic detail.

6 An extended example: JIT-CCS

This section provides a complete extended example, with commentary, of how constructs which support orthogonal specification can be used to define an implementation style. The implementation style which is used in this example is Just-in-Time cache compiled (JIT-CCS) simulation [5,20]. The Power PC instruction set is used as the target ISA.

A JIT-CCS simulator is an interpreter which caches the results of decoding – the instruction function and pointers to the operands. The simulator reuses this decoding when it executes the instruction in the future. Thus decode happens only once per static instruction. Self-modifying code is supported by invalidating the cache when an instruction changes. Note that self-modifying code can be fairly common when simulating general-purpose programs; on ISAs such as PowerPC, dynamic linking is accomplished by modifying code during symbol resolution.

6.1 Base ISA description

We begin by providing a brief explanation of the PowerPC ISA description. The entire PowerPC ISA description is 3987 lines of LIS code, and thus cannot be reproduced here. Instead, we give an outline of the description followed by the LIS code which contributes to the definitions of a representative instruction: the "logical AND" instruction. We then discuss the support within the ISA description for implementation styles. The reader is encouraged to consult Appendix A for details of the syntax and semantics of the LIS statements used in the description as well as [21] for the rationale underlying these statements.

6.1.1 ISA description outline

The following is an outline of the PowerPC ISA description:

1. **Instruction step constant definitions.** This portion defines constants which are used to label portions of the instruction semantics.
2. **Header codesection definition.** This portion contains C preprocessor include statements for standard headers which are required by C++ code embedded within the description.
3. **Description codesection definition.** This portion contains Python code which is used by the LSE simulator constructor.
4. **Type definitions.** This portion consistes of several **codesection** statements which contain declarations of simulator data types such as registers, effective addresses, instructions, simulated threads, and simulator instances.
5. **Operand names.** This portion declares names for the source and destination operands as well as the action labels which will decode, read, and write these operands.
6. **Field declarations.** This portion declares the names and types of intermediate values which could be communicated to a timing simulator through an interface.

The following LIS code defines the fields which will be seen in later code examples:

```
1  field Opcode        LSE_emu_opcode_t;
2  field instr         instr_t;
3  field inline_pc     addr_t = branch_targets[0];
4  field fault         PPC_fault_t;
5  field fetchEffAddr  eaddr_t;
6  field is64bit       bool : false;
```

7. **Helper code.** This portion consists of **codesection** statements which contain declarations of simple helper functions which perform actions such as computing new condition codes and extract bits from values.
8. **Operand accessors.** This portion declares how to decode, read, and write instruction operands using **accessor** statements which provide the C++ code for these operations. These declarations of how to access operands are known as *accessors*.
9. **Formatting and decoding.** This portion declares the instruction formats and how to map bit patterns to instructions.
10. **Instruction semantics.** This portion declares the semantics of instructions. In order to reduce the size of the description, common behavior is factored out into instruction classes which are used as base classes for the instructions.

6.1.2 Describing the logical AND instruction

The PowerPC logical AND instruction has two source and destination operands, reads from the XER register, and writes to the CR register. Its decoding is described using the following lines of LIS code:

```
1  instrclass commonPPCformat {
2    format OPCD[31:26], XO[10:1], rA[20:16], rB[15:11], rS[25:21], Rc[0];
3    classes ALLonce;
4  }
5
6  instructionlist commonPPCformat [ and ];
7
8  match and      OPCD=31, XO=28;
```

Line 2 declares bitfield names which are used in the PowerPC manual [1]. (There are actually many more bitfields defined which are used by other instructions, but they have been removed for reasons of space.) The bitfield definitions are placed in an instruction class *commonPPCformat* which will be used as a base class. This class in turn inherits (line 3) from another class *ALLonce*, which will be shown later.

Line 6 declares a list of instructions which inherit from *commonPPCformat*. In the full description, all instructions are listed here.

Line 8 declares that the *and* instruction should be recognized when the *OPCD* field is 31 and the *XO* field is 28.

The semantics of the logical AND instruction are almost completely inherited from a number of instruction classes which hold common instruction behavior. The following LIS code declares all of these classes:

```
1   instrclass ALLonce {
2
3     action @ clearStep = {
4       fault = no_fault;
5     }
6
7     action @ clearAllStep = {
8       /* clear all the information */
9       memset(&LIS_ii,0,sizeof(LIS_ii));
10
11      /* set valid fields */
12      LIS_ii.addr = a;
13      LIS_ii.swcontexttok = swtok;
14      LIS_ii.hwcontextno = hwcno;
15    }
16
17    action @ fetchStep = {
18      size = 4;
19      void *ha = ctx.softmmu.translate(addr & ~0x3, 0, 4, softmmu_t::Ifetch,
20                                       MMUhandler(ctx, fault, fetchEffAddr));
21      instr = fault ? 0 : LSE_b2h(*reinterpret_cast<uint32_t *>(ha));
22    };
23
24    action + requiredDecodeStep = {
25      branch_dir = 0;
26      inline_pc = addr + 4;
27    }
28
29    action @ reportOpcodeStep = { Opcode = LIS_opcode; };
30  }
31
32  instrclass standardbehavior {
33    action @ calcNPCStep = { next_pc = inline_pc; };
34  }
35
36  instructionlist standardbehavior [ and stw ];
37
38  instrclass r3_nOE_rC_1 {
39    operand src1 R(rS);
40    operand src2 R(rB);
41    operand src3 XERSO(Rc);
42    operand dest1 R(rA, !fault);
43    operand destCR CRfield(0, 0xf, Rc, !fault);
44  }
45
46  classes and r3_nOE_rC_1;
47  action and @ evaluateStep = {
48    dest1 = src1 & src2;
49    destCR = Impl::calcNewCR0(dest1, src3);
50  };
```

Lines 1–30 define the *ALLonce* instruction class, which contains semantics which are common across all instructions. We have previously seen that the logical AND instruction inherits indirectly from this class. The common semantics include clearing of various instruction fields before execution, instruction fetch, computing the inline PC, and reporting the opcode. The address translation on lines 19–20 interacts with implementation styles and will be discussed further in Section 6.1.3.

Lines 32–34 define the *standardbehavior* class, which simply holds semantics that set the next PC to the inline PC. Line 36 declares that the *and* instruction inherits from this class.

Lines 38–44 define the operand read and write semantics for all instructions which read two general purpose registers and the XER register and which write one general purpose register and the CR register. The **operand** statements declare that the instruction class has an operand with a particular name; this name must have been previously defined with an **operandname** statement. The **operand** statement also declares which accessor (in this example, *R*, XERS0, or *CRfield*) should be used to read or write the operand and what parameters to pass to the accessor. Line 46 declares that the *and* instruction inherits these operands.

Finally, lines 47–49 declare the semantics which are unique to the logical AND. calcNewCR0 is a helper function declared within a **codesection** at an earlier point in the ISA description file.

Taken all together, the complete semantics for the AND instruction are:

```
fault = no_fault;

size = 4;
void *ha = ctx.softmmu.translate(addr & ~0x3, 0, 4, softmmu_t::Ifetch,
                                 MMUhandler(ctx, fault, fetchEffAddr));
instr = fault ? 0 : LSE_b2h(*reinterpret_cast<uint32_t *>(ha));

decodetoken = decode(instr); // inserted automatically

branch_dir = 0;
inline_pc = addr + 4;

Opcode = LIS_opcode;

src1 = R::read(ctx, rS);
src2 = R::read(ctx, rB);
src3 = XERS0::read(ctx, rC);

dest1 = src1 & src2;
destCR = Impl::calcNewCR0(is64bit, dest1, src3);

next_pc = inline_pc;

R::write(ctx, dest1, rA, !fault);
CRfield::write(ctx, destCR, 0, 0xf, Rc, !fault);
```

The accessors which are used by this instruction simply read and write particular registers and are declared in the following fashion:

```
1   accessor regR_t rval = R(int regno, bool dowrite=true) {
2     decode = {
3       oi.spaceid = LSE_emu_spaceid_GR;
4       oi.spaceaddr.GR = regno;
5       oi.uses.reg.bits[0] = (1<<regno);
6     };
7     read = { return ctx.r[regno]; };
8     write = { if (dowrite) ctx.r[regno] = data; };
9   }
10
11  accessor regR_t rval = XERSO(bool really=true) {
12    decode = {
13      if (really) {
14        oi.spaceid = LSE_emu_spaceid_SPR;
15        oi.spaceaddr.SPR = PPC_SPR_XER;
16        oi.uses.reg.bits[0] = xer_so_mask;
```

```
17        }
18      };
19      read = { return ctx.xer; };
20    }
21
22    accessor uint32_t cval = CRfield(int regno, uint32_t mask=0xf,
23                                     bool pred=true, bool dowrite=true) {
24      decode = {
25        if (pred) {
26          oi.spaceid = LSE_emu_spaceid_OUR;
27          oi.spaceaddr.OUR = PPC_OUR_CR;
28          oi.uses.reg.bits[0] = mask << (28 - 4*regno);
29        }
30      };
31      read = { return (ctx.cr >> (28 - 4*regno)) & 0xf; };
32      write = {
33        uint32_t rmask = mask << (28 - 4*regno);
34        if (pred && dowrite)
35          ctx.cr = ctx.cr & ~rmask | (data << (28 - 4*regno)) & rmask;
36      };
37    }
```

6.1.3 ISA support for implementation styles

Very little support is needed within the ISA description to support implementation styles. This support falls into two categories: data structure extension hooks and software MMU hooks. Data structure extension hooks simply allow a data structure to be extended via **codesection** statements. Software MMU hooks allow behavior to take place when certain events occur during address translation.

The functional simulator instance data structure (PPC_dinst_t) has hooks at initialization, finalization, argument parsing, and command-line usage printing. These hooks are generally used for user-settable parameters as well as code caches which are shared between all simulated threads. The LIS code is shown below; the hooks consist of **LIS_CODESECTION** declarations at lines 13, 25, 29, 35, and 40 within two of the standard **codesection** statements.

```
1     codesection private {
2
3       class PPC_dinst_t {
4       public:
5         LSE_emu_interface_t *einterface;
6         ... other fields ...
7
8         PPC_dinst_t(LSE_emu_interface_t *i);
9         ~PPC_dinst_t();
10        int parse_arg(int argc, char *arg, char *argv[]);
11        void print_usage();
12
13        LIS_CODESECTION(interfaceFields)
14      };
15    }
16
17    codesection support {
18
19      PPC_dinst_t::PPC_dinst_t(LSE_emu_interface_t *i) :
20      einterface(i), EMU_emulateOS(0), EMU_trace_syscalls(0),
21        EMU_debugcontextload(0)
22          {
```

```
23          evars = new LSE_env::env_t();
24          ostoken = 0;
25          LIS_CODESECTION(initializeInterfaceFields);
26        }
27
28    PPC_dinst_t::~PPC_dinst_t() {
29      LIS_CODESECTION(finalizeInterfaceFields);
30      delete evars;
31    }
32
33    int PPC_dinst_t::parse_arg(int argc, char *arg, char *argv[]) {
34      int rval = 0;
35      LIS_CODESECTION(parse_arg);
36      return rval;
37    }
38
39    void PPC_dinst_t::print_usage() {
40      LIS_CODESECTION(print_usage);
41    }
42  }
```

The simulation thread data structure (PPC_context_t) is not shown here; it has four **LIS_CODESECTION** hooks: one for declaring additional fields, one for initialization, one for finalization, and one for copying. These hooks are generally used for per-thread code caches.

The software MMU is used in the PowerPC description to cache translations from target virtual addresses to target physical addresses and to host virtual addresses. When a virtual to physical/host translation is required (e.g., when doing instruction fetch), the software MMU is consulted. On a hit, the physical and host addresses are simply returned. On a miss, the operating system emulator is consulted to determine the physical address, and then the physical address is translated to a host address. The software MMU maintains write, read, and execute permission bits on a page granularity to allow the functional simulator to efficiently report data and instruction access errors.

There are three software MMU hooks which support implementation styles: a miss hook called on MMU miss, a clear hook called when the MMU is cleared (such as when the operating system emulator changes mappings), and a dropped mapping hook called when a virtual page is unmapped by the operating system emulation. These hooks are generally used to keep code caches coherent. The placement of these hooks is shown below:

```
1   codesection support {
2     bool MMUhandler::miss(addr_t va, unsigned int mode,
3                           softmmu_t::kinds kind, entry& ent) const {
4       void *ha;
5       int mask = do_translate(va, kind, ha, ent.extra);
6
7       ... update the soft MMU
8
9       // any special stuff that implementations need (e.g. code caches)
10      LIS_CODESECTION(softmmu_miss_hook);
11
12      return !!mask;
13    }
14
15    void MMUhandler::clear() const {
16      LIS_CODESECTION(softmmu_clear_hook);
```

```
17       }
18
19       void PPC_drop_mapping(PPC_dinst_t *di, LSE_emu_addr_t va,
20                             LSE_emu_addr_t pa) {
21         LIS_CODESECTION(drop_mapping_hook);
22       }
23     }
```

Note that both the data structure extensions and the software MMU use hooks based on the **LIS_CODESECTION** statement to defer behavior to the implementation styles. This construct is not the only possible way of deferring behavior; subclasses of the data structures and/or calls to functions which the style must implement would have also worked. We prefer to use hooks because: 1) they cost nothing at runtime if the implementation style does not need them, 2) the text filling the hook remains extensible even after an implementation style is applied, and 3) implementation styles which do not use a hook do not need to implement it.

6.2 JIT-CCS implementation

The JIT-CCS implementation style needs to:

- Modify instruction behavior to use the operand pointers
- Decode and cache the results when a static instruction is first seen
- Look up and use the cached results on later dynamic invocations of the static instruction.
- Invalidate the cache when code is modified.

Our implementation of JIT-CCS uses three LIS files: a register pointer description, a JIT-CCS description, and a code cache description.

Our implementation of JIT-CCS differs from that of [20] in the way that cache invalidations are handled. [20] re-reads the instruction word on each instruction execution and compares it against the cached instruction word to determine whether the instruction has changed. We invalidate all cached information for instructions in a page of simulated target memory when a page is written to, thus saving the instruction fetch on every instruction execution.

6.2.1 Register pointer description

We start with the description of register pointers. This file uses the openness of the LIS constructs for instructions and instruction classes to add semantics for generating and using register pointers to all the instructions.

```
1    buildset CACHE_REGPTRS ALLonce {
2
3      structfield CACHE_REGPTRS_ptrs long regPtr1;
4      structfield CACHE_REGPTRS_ptrs long regPtr2;
5      structfield CACHE_REGPTRS_ptrs long regPtr3;
6      structfield CACHE_REGPTRS_ptrs long regPtr5;
7      structfield CACHE_REGPTRS_ptrs long regPtr6;
8      typedef CACHE_REGPTRS_ptrs *CACHE_REGPTRS_ptrsP_t;
```

```
 9     field regptrsP CACHE_REGPTRS_ptrsP_t;
10
11     hide &regptrsP, &instr, &decodetoken, &swcontexttok;
12     entrypoint void
13       CACHE_REGPTRS_set_regptrs(void *swcontexttok,
14                                 LSE_emu_decodetoken_t decodetoken,
15                                 instr_t instr,
16                                 CACHE_REGPTRS_ptrsP_t __restrict__ regptrsP)
17       = { decodetoken } 2000000;
18
19     action r3_nOE_rC_1 @ fetchOp1Step-4 = {
20   #ifdef CACHE_REGPTRS_GUARD
21     if (1) src1 = *(regR_t *)regptrsP->regPtr1;
22     else
23   #endif
24     }
25
26     action r3_nOE_rC_1 @ fetchOp2Step-4 = {
27   #ifdef CACHE_REGPTRS_GUARD
28     if (1) src2 = *(regR_t *)regptrsP->regPtr2;
29     else
30   #endif
31     }
32
33     action r3_nOE_rC_1 @ writeOp1Step-4 = {
34   #ifdef CACHE_REGPTRS_GUARD
35     if (1) {
36       if (LIS_dospeculation && !(LIS_ii.LIS_operand_backed & 1))
37           LIS_ii.LIS_operand_rb[0].rval = *(regR_t *)regptrsP->regPtr5;
38       if (!fault) *(regR_t *)regptrsP->regPtr5 = dest1;
39     } else
40   #endif
41     }
42
43     action r3_nOE_rC_1 +2000000 = {
44       regptrsP->regPtr1 = (long)&ctx.r[rS];
45       regptrsP->regPtr2 = (long)&ctx.r[rB];
46       regptrsP->regPtr5 = (long)&ctx.r[rA];
47     }
48
49     ...
50
51   } // buildset CACHE_REGPTRS
52
53   buildset cache_with_regptrs ALLonce {
54     implement CACHE_REGPTRS = single;
55     implement codecache = single;
56     codesection codecache_entries {
57       CACHE_REGPTRS_ptrs regptrs;
58     }
59   }
```

The register pointer handling is defined inside of a **buildset** statement, which normally is used to define an interface. Here it is being used as a kind of "optional library" wrapper; the buildset has no implementation style given and thus will not take effect unless some other portion of LIS code assigns an implementation style to it.

Lines 3-9 define a datatype for holding pointers to registers and then create a field in the instruction information structure to hold those pointers.

Lines 19-46 describe how to use register pointers for one class of instructions. The class is named *r3_nOE_rC_1*, and was introduced in Section 6.1.2 as the class of all

PowerPC instructions which have two integer source registers, one integer destination register, read the XER register, and update the CR register.

Lines 19-24 and lines 26-31 define code which is inserted just before the normal operand fetch code. This code stores the results of dereferencing the register pointer into the appropriate source operand. The code is predicated with an always-true predicate so that the else clause may predicate the normal function call which is made to read an operand. The new code is enclosed in a preprocessor guard; this guard is used because this code is added for *all* implementation styles, but we only want to use it in the ones which have register pointers. Those implementation styles will define the guarding macro. The resulting instruction function for the AND instruction in the operand fetch interface function of the **Steps** interface is:

```
void EMU_doStpAllN3_token_add(LSE_emu_instr_info_t &LIS_ii,
                              void *swcontexttok, LSE_emu_iaddr_t addr,
                              instr_t instr) {

  regR_t& src1 LIS_UNUSED = LIS_ii.operand_val_src[1].data.rval;
  regR_t& src2 LIS_UNUSED = LIS_ii.operand_val_src[2].data.rval;
  const unsigned long rA LIS_UNUSED = 0x1fUL & (instr >> 16);
  const unsigned long rB LIS_UNUSED = 0x1fUL & (instr >> 11);

#ifdef CACHE_REGPTRS
  if (1) src1 = *(regR_t *)regptrsP->regPtr1;
  else
#endif
    src1 = LSEemu_accessors::R::read(LIS_ii, LIS_dospeculation , ctx, rA);

#ifdef CACHE_REGPTRS
  if (1) src2 = *(regR_t *)regptrsP->regPtr2;
  else
#endif
    src2 = LSEemu_accessors::R::read(LIS_ii, LIS_dospeculation , ctx, rB);

} /* EMU_doStpAllN3_token_add */
```

Lines 33-41 define similar code which is inserted just before the normal operand writeback code. This code is slightly more complex because it must backup the current value of the destination register when the instruction is being executed speculatively (lines 36-37) and it must not update the register if there has been a fault in instruction execution.

Lines 43-47 define code which assigns values to the register pointers. The pointers are made to point to the appropriate register in the simulated thread data structure. This code is assigned to a step named 2000000; this number is chosen to be well out of the range of names of the normal instruction semantics in the PowerPC ISA description.

Of course, lines 19-47 define just one class of instructions. Similar code, tedious to write and look at, but easy to create, exists for the rest of the instruction classes.

Lines 12-17 define an interface function which collects the semantics at step 2000000 for all the instructions into one function. Line 11 ensures that interface function's parameters are used instead of instruction fields where the names match. When the *single* implementation style is used to implement the buildset, the generated code for this interface function looks like:

```
1  void CACHE_REGPTRS_set_regptrs(LSE_emu_instr_info_t &LIS_ii,
2                                 void *swcontexttok,
3                                 LSE_emu_decodetoken_t decodetoken,
4                                 instr_t instr,
5                                 CACHE_REGPTRS_ptrsP_t regptrsP) {
6
7    LSE_emu_isacontext_t &ctx LIS_UNUSED
8      = *(static_cast<LSE_emu_isacontext_t *>(swcontexttok));
9
10   switch ( decodetoken ) {
11
12   case LSE_emu_decodetoken_and:
13   case LSE_emu_decodetoken_andc:
14   case LSE_emu_decodetoken_eqv:
15   case LSE_emu_decodetoken_nand:
16   case LSE_emu_decodetoken_nor:
17   case LSE_emu_decodetoken_or:
18   case LSE_emu_decodetoken_orc:
19   case LSE_emu_decodetoken_sld:
20   case LSE_emu_decodetoken_slw:
21   case LSE_emu_decodetoken_srad:
22   case LSE_emu_decodetoken_sraw:
23   case LSE_emu_decodetoken_srd:
24   case LSE_emu_decodetoken_srw:
25   case LSE_emu_decodetoken_xor:
26     {
27       const unsigned long rA LIS_UNUSED = 0x1fUL & (instr >> 16);
28       const unsigned long rB LIS_UNUSED = 0x1fUL & (instr >> 11);
29       const unsigned long rS LIS_UNUSED = 0x1fUL & (instr >> 21);
30
31       regptrsP->regPtr1 = (long)&ctx.r[rS];
32       regptrsP->regPtr2 = (long)&ctx.r[rB];
33       regptrsP->regPtr5 = (long)&ctx.r[rA];
34
35     }
36     break;
37
38     ... // many more cases
39   }
40 }
```

6.2.2 JIT-CCS description

We now explain the LIS code which describes the JIT-CCS implementation style itself.

```
1  style JIT_CCS {
2
3    codesection JIT_CCS_headers {
4  #define CACHE_REGPTRS_GUARD
5    }
6
7    implement cache_with_regptrs = single;
8
9    specialize instr, addr, regptrsP;
10
11   generator latepublic {
12     void %%NAME() (%%PARMS());
13   }
14
15   codesection codecache_entries {
16     JIT_CCS::LIS_atable_t jitccs_table;
```

```
17      }
18
19      generator JIT_CCS_epilogue {
20        void %%NAME()(%%PARMS()) {
21          %%DECLS();
22
23        tryagain:
24          codecache_ent *lb = codecache_vlookup(ctx, addr, true);
25
26          if (!lb) {
27            %%BEFORE();
28
29            CACHE_REGPTRS_ptrs regptrs;
30            regptrsP = &regptrs;
31            CACHE_REGPTRS_set_regptrs(LIS_ii, swcontexttok, decodetoken, instr,
32                                      regptrsP);
33
34            %%IFAFTER(%%RTYPE() (*ep)(%%AFTERPARMS()) = %%AFTERPTR();
35                      ep(%%AFTERARGS()););
36            return;
37          }
38
39          JIT_CCS::LIS_atable_t *ep = &lb->code.d.jitccs_table;
40          if (!ep->%%NAME()) {
41
42            if (ctx.di->codecache->bytesUsed / 1024 >= ctx.di->cache_size) {
43              ctx.di->codecache->reset(true); // hook?
44              goto tryagain;
45            }
46
47            %%BEFORE();
48            lb->code.d.jitccs_table = *(ep = & %%AFTERREC());
49            lb->code.d.instr = instr;
50            lb->code.d.decodetoken = decodetoken;
51            regptrsP = &lb->code.d.regptrs;
52            CACHE_REGPTRS_set_regptrs(LIS_ii, swcontexttok, decodetoken, instr,
53                                      regptrsP);
54          } else {
55            instr = lb->code.d.instr;
56            regptrsP = &lb->code.d.regptrs;
57            decodetoken = lb->code.d.decodetoken;
58          }
59
60          return (*ep->%%NAME())(%%AFTERARGS());
61        } // %%NAME()
62      } // JIT_CCS_epilogue
63
64    } // style JIT_CCS
```

Setting up to use register pointers and code caches

Lines 3–5 add a definition of the CACHE_REGPTRS_GUARD preprocessor macro to the file which contains JIT-CCS-implemented interface and instruction functions. As was explained in Section 6.2.1, this macro is the guard for the code inside instruction functions which uses register pointers.

Line 9 declares that the instruction encoding, address, and register pointers should be added to the signature of each instruction function.

Lines 15–17 add a table of type LIS_atable_t with pointers to the instruction functions into the codecache entry structure. This datatype is generated automatically by the tool chain.

Line 7 implements a buildset which combines the code cache and register pointer buildsets, causing both code cache and register code to be instantiated. It also adds the register pointers to the definition of a code cache entry. The definition of this buildset is:

```
1  buildset cache_with_regptrs ALLonce {
2    implement CACHE_REGPTRS = single;
3    implement codecache = single;
4    codesection codecache_entries {
5      CACHE_REGPTRS_ptrs regptrs;
6    }
7  }
```

The interface functions

Lines 11–13 of the JIT-CCS description declare the signature for each interface function implemented using this style in the public header file for the functional simulator so that the interface function may be called by clients of the simulator. The **generator** statement generates the enclosed code for each interface function; each interface function's name and parameters are supplied via the **%%NAME()** and **%%PARMS()** macros which query the interface description.

The bulk of the JIT-CCS style's description is found in lines 19–62, which generate the interface functions. The **%%DECLS()** macro inserts code which defines any instruction fields used within the code generated by the **generator** statement. For example, the first part of the generated code for the **Minimal** interface in the experiment of Section 5 looks like:

```
void EMU_doOneMinN(LSE_emu_instr_info_t & LIS_ii, void * swcontexttok) {

    LSE_emu_iaddr_t &addr = LIS_ii.addr;
    LSE_emu_isacontext_t &ctx
            = *(static_cast<LSE_emu_isacontext_t *>(swcontexttok));
    LSE_emu_addr_t size;
    instr_t &instr = LIS_ii.instr;
    PPC_fault_t &fault = LIS_ii.fault;
    LSE_emu_decodetoken_t &decodetoken = LIS_ii.decodetoken;
    JIT_CCS_ptrsP_t regptrsP ;
```

Some instruction fields (e.g., *fault*) are made to refer to the instruction information structure (LIS_ii) while others (*regptrsP*) are local variables. The selection of which to use is controlled by the visibility attributes of the interface (Appendix A.6.3). Note that fields such as size are not explicitly in the JIT-CCS description, but are inserted because the **%%BEFORE()** macro does generate code which references them after querying the instruction descriptions.

Line 24 looks up the entry for the instruction's virtual address in the code cache. The address is in the addr field of the instruction and the ctx field refers to the simulated thread in which an instruction is executing. The cache lookup returns a pointer to a cache entry. If the cache missed, the lookup routine allocates a new entry as long as the virtual address can be successfully translated to a physical address.

If the cache lookup finds that the virtual address cannot be translated, lines 26–37 interpret the instruction, correctly reporting an instruction fault. The **%%BEFORE()** macro on line 27 inserts the semantics of the instruction up to and including decoding. Lines 34–35 perform the semantics of the function after decoding by inserting a call to the appropriate instruction function. The **%%IFAFTER** macro is used to ensure that this call only is inserted if the interface actually included post-decode semantics. Because the instruction have had the use of operand pointers inserted, these pointers must be properly initialized; this happens in lines 29–32, where the *CACHE_REGPTRS_set_regptrs* interface function defined in Section 6.2.1 is called. Lines 26–37 generate the following code:

```
if (!lb) {
  fault = no_fault;
  size = 4;
  eaddr_t dummyPA;
  void *ha = ctx.softmmu.translate(addr & ~0x3, 0, 4, softmmu_t::Ifetch,
                                   MMUhandler(ctx, fault, dummyPA));
  instr = fault ? 0 : LSE_b2h(*reinterpret_cast<uint32_t *>(ha));
  decodetoken = do_standard_decoding(LIS_ii, instr);

  CACHE_REGPTRS_ptrs regptrs;
  regptrsP = &regptrs;
  CACHE_REGPTRS_set_regptrs(LIS_ii, swcontexttok, decodetoken, instr,
                            regptrsP);

  void (*ep)(LSE_emu_instr_info_t &__restrict__ LIS_ii,
             LSE_emu_iaddr_t addr, instr_t instr,
             CACHE_REGPTRS_ptrsP_t regptrsP) =
    (JIT_CCS::LIS_atable[ decodetoken ].EMU_dofast);

  ep(LIS_ii, addr, instr, regptrsP);
  return;
}
```

When the cache lookup is successful, line 40 checks to see whether the cache entry contains any information. If it does, lines 55–57 copy the cached information into the instruction fields.

If the entry does not contain information, lines 47–52 compute and fill in that information. Line 47 inserts instruction semantics up to and including decoding. Lines 48–52 initialize the values in the cache. Lines 42–44 bound the size of the cache; when it gets too large, the cache is cleared.

Finally, on line 60, the instruction function which contains the instruction behavior after decoding is called, based upon the cached pointer to that function. Lines 40–61 generate:

```
if (!ep->EMU_dofast) {
  if (ctx.di->codecache->bytesUsed / 1024 >= ctx.di->cache_size) {
    ctx.di->codecache->reset(true); // hook?
    goto tryagain;
  }

  fault = no_fault;
  size = 4;
  eaddr_t dummyPA;
  void *ha = ctx.softmmu.translate(addr & ~0x3, 0, 4, softmmu_t::Ifetch,
```

```
                                     MMUhandler(ctx, fault,dummyPA));
    instr = fault ? 0 : LSE_b2h(*reinterpret_cast<uint32_t *>(ha));
    decodetoken = do_standard_decoding(LIS_ii, instr);

    lb->code.d.jitccs_table = *(ep = & (JIT_CCS::LIS_atable[ decodetoken ]));
    lb->code.d.instr = instr;
    lb->code.d.decodetoken = decodetoken;
    regptrsP = &lb->code.d.regptrs;
    CACHE_REGPTRS_set_regptrs(LIS_ii, swcontexttok, decodetoken, instr,
                              regptrsP);
  } else {
    instr = lb->code.d.instr;
    regptrsP = &lb->code.d.regptrs;
    decodetoken = lb->code.d.decodetoken;
  }

  return (*ep->EMU_dofast)(LIS_ii, addr, instr, regptrsP);
```

6.2.3 Code cache

Finally, we describe the code cache. We do not show the complete code for the code cache, as it runs to several hundred lines of code, but describe its operation and show highlights from the code illustrating the use of LIS concepts.

Code cache architecture

Our code cache implementation is a two-level cache. First, the virtual address is used to index into a small direct-mapped per-thread cache. This lookup is implemented as an inline function in the private header file so that it may be inlined and optimized:

```
1  codesection private {
2    inline codecache_ent *codecache_vlookup(LSE_emu_isacontext_t &ctx,
3                                            LSE_emu_addr_t va,
4                                            bool perContext = false) {
5      unsigned int idx = (va >> 2) % codecache_vcache_size;
6      codecache_ent *lb = static_cast<codecache_ent *>(ctx.vcodecache[idx]);
7      if (lb->pc.va != va)
8        return ctx.di->codecache->vlookup_failed(ctx, va, perContext);
9      else return lb;
10   }
11 }
```

If the address is not found in the virtual cache, we perform virtual-to-physical address translation and use the combination of virtual address and physical address to look for it in the second-level physical cache. There are two choices of second-level cache to use: a per-thread or a shared cache. JIT-CCS uses the per-thread cache since the register pointers point directly into thread data structures and thus cannot be shared between threads. Other clients of the code cache implementation, such as binary translation, can use the shared cache. This code is not shown as it does not illustrate features of LIS.

LIS is used to declare the virtual code cache and per-thread cache to be added to the simulator's thread data structure and the global cache, as well as initialization and finalization code for them:

```
1  codesection interfaceFields {
2    class codecache_t *codecache;
3  }
4
5  codesection initializeInterfaceFields {
6    codecache = new codecache_t(this);
7  }
8
9  codesection finalizeInterfaceFields {
10   delete codecache;
11   codecache = 0;
12 }
13
14 codesection contextFields {
15   void *vcodecache[codecache_vcache_size];
16   class pcodecache_t *codecache;
17 }
18
19 codesection contextCopyHook {
20   codecache = new pcodecache_t;
21 }
22
23 codesection contextDelete {
24   codecache->reset(true, codecache_ctx_remove_hook(di, this));
25   delete codecache;
26   codecache = 0;
27 }
```

The first three codesections are for the global cache; the final three are for the per-thread caches. Note that the codesection names are *not* standard, tool-chain-enforced names; rather the PowerPC ISA description declare these codesections as "hooks" as described in Section 6.1.3 to permit extensibility. The addition of hooks is the only way in which the instruction set description has ever needed to be changed to support new implementation styles; such a small change requires little revalidation – no more than a quick sanity check – as an empty hook generates no new C++ code.

Entries in the code cache are invalidated whenever a write occurs to the same target memory physical page that the entries come from. The tracking for invalidation is done similarly to QEMU [6] in that the PowerPC description of how to translate virtual addresses to physical addresses uses a software TLB to cache translations; when a software TLB miss occurs, invalidation occurs. In QEMU, this behavior is hard-coded; in our LIS-based simulators, the software TLB code in the PowerPC description uses several codesections as hooks which are then filled by the code cache code:

```
1  codesection softmmu_miss_hook {
2    if ((mask & 4) && kind == softmmu_t::Store) {
3      unsigned char *pp = ctx.di->get_pattr_ptr(pa);
4      if ((*pp) & 1) {
5        ctx.di->codecache->remove_information(pa);
6        *pp &= ~1;
7      }
8    }
```

```
 9    }
10
11    codesection softmmu_clear_hook {
12      ctx.di->codecache->clear_vcodecache(&ctx);
13    }
14
15    codesection drop_mapping_hook {
16      di->codecache->remove_information(pa);
17    }
```

The above listing also includes hooks executed when the software TLB is cleared and when a virtual-to-physical address mapping is dropped; in either case cache entries should be invalidated.

In line 2 of the above listing, mask indicates the type of access which has missed in the software TLB. Line 5 looks up a flag byte kept on a per-physical-page basis (pa gives the physical address). This flag indicates whether the code cache has cached any information on that page. If it has, then the matching information is removed and the flag is cleared. The flag is set when the code cache allocates an entry; at the same time, all software TLBs remove their entries for the page or mark the page as non-writable.

Interactions with styles

An implementation style interacts with the code cache in four ways. First, it must add fields to the code cache entries to store information it finds useful. Section 6.2.2 showed fields such as register information, address, and decode results being added by filling in the *codecache_entries* codesection. Second, after looking up an entry in the cache, it must use the resulting fields. Third, when a new entry is allocated, it must fill in the fields. These latter two requirements are met within the interface function and were described previously for JIT-CCS. Finally, when a code cache entry is deallocated, the style may need to clean up some data. JIT-CCS does not need to do any cleanup, but binary translation does, for example – it must reclaim memory allocated to generated code. Styles which need to clean up fill in the *codecache_remove_hook* codesection.

6.3 Thoughts

The code cache, implementation style, and ISA description must all cooperate to allow a caching implementation style to work. This cooperation is made possible without seriously compromising the orthogonality principle by the placement of only a handful of hooks in the ISA description.

It could be argued that the interaction between the software TLBs and the code cache is a compromise of the orthogonality principle. In particular, the need for the code cache to have a software TLB available to it and to be able to invalidate its entries does constrain the ISA description. However, software TLBs are a good idea for improving the performance of address translation and are likely to be present anyway; indeed, our operating system emulation code requires them. Once they are

present, adding codesections as hooks is a non-intrusive operation which is not likely to introduce new bugs into old style and interface descriptions – the new codesections would remain unfilled in old styles and/or interfaces, and thus generate no new code.

The argument that the ISA is not seriously affected by the implementation style cannot be reversed: the implementation style can be significantly affected by the ISA. For example, the JIT-CCS style needs to provide register pointer code for each instruction format. However, once an implementation style has been adapted for a particular ISA, no further work is necessary to support multiple interfaces implemented using that style.

7 Conclusions

We have introduced the Orthogonal-Specification Principle for functional simulators and described how ADLs can be extended to enable orthogonal specification of implementation styles. No previous ADL has provided a set of constructs which support orthogonal specification. We have shown through a case study that using an ADL with these extensions enables rapid development and application of implementation styles to multiple interfaces.

As a result of this work, architects will be able to easily specify implementation styles and apply them to create simulators with multiple, evolving interfaces – thus improving both simulator speed and development time and allowing exploration of a greater portion of the design space with longer multi-threaded benchmarks, leading to improved designs.

8 Availability

Source code for LIS can be downloaded as part of the Liberty Simulation Environment at `http://bardd.ee.byu.edu/`. Descriptions are provided for the PowerPC and SPARC instruction sets with interpretive simulation and the PowerPC JIT-CCS implementation style. The SPARC instruction set is capable of full-system simulation. Descriptions for other implementation styles (e.g. binary translation) are available by contacting the authors.

Acknowledgements This work was supported by National Science Foundation grant CCF-1017004. We would also like to thank the reviewers for their very helpful feedback.

References

1. PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. G522-0290-01. International Business Machines Corporation (2000)
2. Alameldeen, A.R., Wood, D.A.: Variability in architectual simulations of multi-threaded workloads. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture (2003)
3. Amicel, R., Bodin, F.: Mastering startup costs in assembler-based compiled instruction-set simulation. In: Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (2002)

4. Bartholomeu, M., Azevedo, R., Rigo, S., Araujo, G.: Optimizations for compiled simulation using instruction type information. In: Proceedings of the 16th Symposium on Computer Architecture and High-Performance Computing, pp. 74–81 (2004)
5. Bedichek, R.: Some efficient hardware simulation techniques. In: Proceedings of the USENIX Winter 1990 Technical Conference, pp. 53–63 (1990)
6. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the 2005 USENIX Annual Technical Conference, pp. 41–46 (2005)
7. Chiou, D., Angepat, H., Patil, N.A., Sunwoo, D.: Accurate functional-first multicore simulators. IEEE Computer Architecture Letters **8**(2), 64–67 (2009)
8. Chiou, D., Sunwoo, D., Kim, J., Patil, N., Reinhart, W.H., Johnson, D.E., Xu, Z.: The FAST methodology for high-speed SoC/computer simulation. In: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design, pp. 295–302 (2007)
9. Chung, M.K., Kyung, C.M.: Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time. In: Proceedings of the 15th IEEE Workshop on Rapid System Prototyping (RSP '04), pp. 38–44 (2004)
10. Cmelik, B., Keppel, D.: Shade: A fast instruction-set simulator for execution profiling. In: Proceedings of the 1994 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 128–137 (1994)
11. D'Errico, J., Qin, W.: Constructing portable compiled instruction-set simulators – an ADL-driven approach. In: 2006 Conference on Design, Automation and Test in Europe, pp. 112–117 (2006)
12. Jones, D., Topham, N.: High speed CPU simulation using LTU dynamic binary translation. In: Proceedings of the 2009 International Conference on High-Performance Embedded Architectures and Compilers, *LNSC*, vol. 5409, pp. 50–64 (2009)
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), *LNCS*, vol. 1241, pp. 220–242 (1997)
14. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 75–86 (2004)
15. Leupers, R., Elste, J., Landwehr, B.: Generation of interpretive and compiled instruction set simulators. In: Proceedings of the Asian-Pacific Design Automation Conference, pp. 339–342 (1999)
16. Mauer, C.J., Hill, M.D., Wood, D.A.: Full-system timing-first simulation. In: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 108–116 (2002)
17. May, C.: Mimic: A fast system/370 simulator. In: Papers of the Symposium on Interpreters and Interpretive Techniques (1987)
18. Mills, C., Ahalt, S., Fowler, J.: Compiled instruction set simulation. Software – Practice and Experience **21**(8), 877–889 (1991)
19. Mong, W.S., Zhu, J.: DynamoSim: A trace-based dynamically compiled instruction set simulator. In: Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design, pp. 131–136 (2004)
20. Nohl, A., Braun, G., Hoffmann, A., Schliesbusch, O., Leupers, R., Meyr, H.: A universal technique for fast and flexible instruction-set architecture simulation. In: Proceedings of the 39th ACM/IEEE Design Automation Conference, pp. 22–27 (2002)
21. Penry, D.A.: A single-specification principle for functional-to-timing simulator interface design. In: Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 186–196 (2011)
22. Penry, D.A., Cahill, K.D.: ADL-based specification of implementation styles for functional simulators. In: Proceedings of the 2011 International Symposium on Systems, Architectures, Modeling, and Simulation., pp. 165–173 (2011)
23. Qin, W., D'Errico, J., Zhu, X.: A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In: Proceedings of the 3rd Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS), pp. 193–198 (2006)
24. Reshadi, M., Dutt, N.: Reducing compilation time overhead in compiled simulators. In: Proceedings of the 21st International Conference on Computer Design, pp. 151–153 (2003)
25. Reshadi, M., Mishra, P., Dutt, N.: Instruction-set compiled simulation: A technique for fast and flexible instruction set simulation. In: Proceedings of the 40th Design Automation Conference (DAC), pp. 758–763 (2003)

26. Sunwoo, D., Kim, J., Chiou, D.: QUICK: A flexible full-system functional model. In: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 249–258 (2009)
27. Vachharajani, M., Vachharajani, N., Penry, D.A., Blome, J.A., August, D.I.: Microarchitectural exploration with Liberty. In: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 271–282 (2002)
28. Witchel, E., Rosenblum, M.: Embra: Fast and flexible machine simulation. In: Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 68–79 (1996). DOI http://doi.acm.org/10.1145/233013.233025
29. Wunderlich, R.E., Wenisch, T.F., Falsafi, B., Hoe, J.C.: SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In: Proceedings of the 30th International Symposium on Computer Architecture, pp. 84–97 (2003)
30. Živojnović, V., Pees, S., Schlager, C., Weber, R., Meyr, H.: SuperSim – A new technique for simulation of programmable DSP architectures. In: Proceedings of the International Conference on Signal Processing Applications and Technology, pp. 1748–1763 (1995)

A The Liberty Instruction Specification Language (LIS)

This appendix gives a brief description of the constructs of the Liberty Instruction Specification Language (LIS). LIS is an architectural description language designed to alleviate the burden of writing functional simulators with multiple interfaces by directly supporting the Single Specification Principle. Using LIS, an simulator developer writes a description of each instruction at a very fine level of granularity and then derives coarser-grained interfaces from the fine-grained interface. Various LIS constructs further simplify the task of writing a functional simulator by allowing common behavior and instruction characteristics to be shared among groups of instructions. The constructs are also designed to allow the instruction set to be easily modified, extended, or overridden. LIS targets the generated functional simulator to interface with the Liberty Simulation Environment (LSE) [27].

Note that style constructs are not described in this appendix because they were described in the main text of the article.

A.1 Basic LIS constructs

A.1.1 Comments and file management

Comments can be introduced into LIS code through either C or C++ style comments. LIS files can be included in other files using the following syntax:

```
include filename
```

A.1.2 Literals and identifiers

Literals in LIS are of two kinds: integers and code. Integer literals are at least 64 bits and can be decimal, binary, octal, or hexadecimal. Code literals begin at any point in the description file where the LIS parser expects such a literal and end when a termination character is found. This character depends upon the LIS statement and is either a semicolon, a right parenthesis, or a closing curly brace. The required terminator is generally clear from the context.

Identifiers begin with an alphabet character or underscore and are followed by alphanumerics and underscores. There are only two name scopes visible at any particular moment in LIS: the global name scope and a local name scope within each buildset or style definition.

A.1.3 Expression Operators

The usual integer, logical, and comparison operators are supported. Comparison and negation operators return 1 for true and 0 for false.

A.1.4 Options and constants

A LIS description can include integer-valued constants and options. The difference between the two is that option values are available in both the generated code and the LIS description while constant values are available only in the LIS description. The syntax used to define them is:

```
constant ident = expr ;
constant ident ?= expr ;
option   ident = expr ;
option   ident ?= expr ;
```

Each form defines a constant or option, but the second and fourth forms will only perform the definition if the constant or option's value has not been previously set. Constant and option values may be changed in LIS descriptions until the point where they are first used.

A.1.5 Control flow

LIS contains conditional constructs, but not loop constructs. The conditional constructs are:

```
if (expr1) {
}
elseif (expr2) {
}
...
else {
}
```

The **elseif** and **else** clauses are optional. Non-zero values of expressions are taken to mean true.

A.1.6 Codesections

The LIS description file can include code to be placed at fixed locations within the generated files. Such code is called a *codesection* and is introduced using the **codesection** statement. This statement has two arguments: the name of the codesection and a piece of C++ or Python code enclosed in curly braces, as shown below:

```
codesection name {
code
}
```

There are a number of standard codesection names which insert code into well-known locations in the generated files. Non-standard codesections may be defined, but they are not included in any of the generated files by default. To include a non-standard codesection into the generated files, place the text **LIS_CODESECTION**(*name*) within a standard codesection; the non-standard codesection will be inserted at that point.

If a codesection is defined more than once, the definitions are concatenated by default. To indicate that a new definition should replace older definitions, prefix the codesection name with a minus (–) sign.

For large codesections defining data types and helper functions which are not expected to change when users extend the instruction set, it may be more convenient to write C++ header files which are simply brought into the code section with #include.

A.1.7 Defining types

Types used in simulator code can be declared and defined in using the **structfield** , **enumvalue**, and **typedef** statements. Types can also be defined directly through codesections. The advantage of the LIS constructs is that the types are constructed in an "open" fashion, allowing the type to be extended easily by later LIS statements, instead of the "closed" fashion required by C++.

The **structfield** statement allows the declaration of fields of a structure. It has the following syntax:

```
structfield structname declaration ;
structfield −structname fieldname ;
```

The first form adds a field to a structure definition; if the structure definition does not exist, it is created. The declaration portion follows the usual C++ field declaration syntax. The second form removes a field from a structure.

The **enumvalue** statement allows the declaration of enumerated types. It has the following syntax:

```
enumvalue typename enumeratedvalue ;
enumvalue typename enumeratedvalue = integervalue ;
enumvalue − typename enumeratedvalue ;
```

The first form adds an enumerated value to an enumerated type; if the enumerated type does not exist, it is created. The second form allows declaration of the integer value to be used to represent the enumerated value; the declaration follows the usual C++ enumerated value syntax. The third form removes an enumerated value from an enumerated type.

The **typedef** statement allows types to be aliased and LIS types to be assigned to simulator types. The syntax is:

```
typedef declaration ;
typedef − typename ;
```

The first form declares a type using the usual C++ typedef syntax. The second form removes a declaration of a type. Note that in LIS multiple definitions of the same type are legal; the last such definition is taken as the correct definition.

LIS-defined type definitions can be made to appear within specific locations within codesections instead of their default location by inserting the text LIS_TYPE(*typename*) into a codesection.

A.2 Operands, state, and intermediate values

A.2.1 Accessing emulated state

Because an extremely common simulator operation is to access emulated state such as registers and memory, LIS supports explicit declaration of accessor methods for state. The syntax of accessor declarations is as follows:

```
accessor datatype fieldname = accessorname ( parameters ) {
   decode|read|write = { C++ code } ;
    ...
}
```

The **accessor** declaration specifies the type of data involved in the access, the field of a LIS-generated union type which should be used to store or source the data, and a name for the accessor. There are five kinds of accessors: decode, read, write, backup and restore. The final two accessors are used to support speculative functional simulation. Accessors have standard parameters which depend upon the kind of accessor; the declaration can add additional parameters. For example, register statespaces usually have register number parameters on their accessors and memory statespaces have address parameters.

A.2.2 Instruction fields

Instruction fields are the fields of a data structure which is filled in for every dynamic instruction instance. They generally contain intermediate values produced during instruction execution, e.g. branch targets. LIS implicitly defines a number of standard instruction fields and provides a means to define additional ISA-specific fields. LIS also uses fields as a way of controlling the granularity of information which the simulator exposes to the user and as storage locations which carry information between different pieces of instruction semantics.

Instruction fields are defined using the following syntax:

```
field fieldname type ;
field fieldname { C++ typedef } ;
field fieldname type = access text ;
```

The first and second forms add a new field. The first form is used when a simple type identifier is sufficient to describe the type of the field; the second form is used when a more complex form type expression (e.g. a pointer to a C++ type) is required. The final form creates an alias to a field or an expression accessing a field; the access text is C++ code which refers to a previously defined field.

A.2.3 Naming operands

LSE simulators store information about values of source and destination operands in arrays within the dynamic instruction instance data structure. These arrays are indexed using operand names. Furthermore, LSE simulators provide the ability to individually fetch source operands and write destination operands using these names. LIS allows the user to easily declare the names using the following syntax:

```
operandname kind index decodeLabel accessLabel = name1 , name2 , ... ;
```

The statement declares the kind of operand (one of src or dest), its index into the appropriate array (which must be non-negative), two action labels, and a list of names for the operand.

Instruction and instruction classes can declare that they have operands through the **operand** attribute (described later); the names of the operands must have been declared through the **operandname** statement. The operand names then become available as references when instruction semantics are defined.

The action labels indicate the action labels at which decoding of the operand will occur and at which reading (for source operands) or writing (for destination operands) will occur when this particular operand name is used by an instruction. It can be helpful to think of the operand names as a list of potential ”times” at which operands can be fetched or written back within the execution of the instruction, with each operand of an instruction being fetched or written back at a different time. Note also that most simulators will attempt to fetch or write back operands in increasing numerical order; choosing names and labels which reflect this behavior can help avoid confusion.

If a particular operand name is defined multiple times, the last definition holds. However, an operand name may not be defined as both a source and a destination operand.

A.3 Defining instructions

Individual instructions within an ISA are defined in LIS by describing their attributes. The most basic syntax for this is the **instruction** statement:

```
instruction instructionname ;
instruction instructionname {
  attribute declarations
}
```

The first form simply declares that an instruction exists. The second form allows the declaration of instruction attributes. If an instruction is defined more than once, the attribute declarations are accumulated. Thus instructions are ”open” objects; they need not be defined all at once. Attributes may be declared outside of an **instruction** statement by inserting the instruction name immediately after the attribute’s keyword.

Each of the attributes will now be described.

A.3.1 Opcode attribute

The opcode attribute sets the opcode for the instruction. The opcode is a string which is used to name the instruction. Opcodes are available to simulators in two ways. The first means is through an enumerated type named LSE_emu_opcode_t which contains identifiers called LSE_emu_opcode_*name*. The second means is through a table of opcode name strings, indexed by LSE_emu_opcode_t. Opcodes are specified with the following syntax:

```
opcode opcodename ;
```

An instruction's opcode is set to its name when the instruction is first defined.

A.3.2 Format attribute

The **format** attribute describes the bit format of an instruction. This format consists of a list of bitfields of the instr instruction field. The syntax is:

```
format bitfieldname [ fromexpr : toexpr ] , ... ;
format bitfieldname [ fromexpr : toexpr ] = expr , ... ;
```

The second form allows *matches*, as described in the next subsection, to be declared with the format. If the instr field is a structure, the bitfield name can be specified in the form: *struct fieldname* : *bitfieldname* .

A.3.3 Match attribute

The **match** attribute specifies the values which the bitfields of an instruction must have in a decoding. The syntax is:

```
match bitfield = expr , ... ;
match - bitfield ;
```

The first form adds match information; bit ranges can also be specified after a bitfield name. Matches can also be specified as a union of matches using the "|" operator. The second form removes match information. The two forms can be combined.

When there are multiple match statements for the same instruction, the match information is computed as the union of the match information from before the statement modified by each match expression in turn. Because a new match statement restricts the already existing matches, additional syntax is needed to extend the union of matches. This syntax is the character +, which indicates "all current matches" and can be used in a union clause:

```
match + | - old match bitfield , new match ;
```

A.3.4 Action attribute

Instruction semantics are specified via *actions*. An action is the finest element of semantic granularity. Actions are grouped together into *entrypoints* to form the code implementing simulator API calls. Entrypoints are discussed in A.6.

The syntax of an action declaration is:

```
action + label-expression = { C++ code }
action @ label-expression = { C++ code }
action - label-expression = { C++ code }
```

Actions are tagged with a non-negative integer label expression; the expression may contain constant integers, symbolic constants, addition, subtraction, and parentheses. Non-negative integers are used for labels to make it simpler to specify ranges of actions when defining entrypoints.

The first form appends the code to any previous action definition at the given label. The second form replaces the definition. The final form replaces only portions of the definition which were not inherited from some parent instruction class (see A.4 for more information on inheritance.)

The code within actions may use instruction field names, bitfield names, operand names (but only for those operands actually defined for the instruction), global options, and options defined in any buildset (see A.6) which uses the action. The variable LIS_opcode holds the decoded opcode in any action taking place after decoding. The variable LIS_ii holds the dynamic instruction information for the instruction. Any information which is to be carried between actions must be stored in instruction fields.

Actions may also contain behavior which is outside of the normal semantics of the instruction; a common example is behavior to disassemble the instruction. By placing the behavior in actions, all of the benefits of instruction manipulation are still possible for the behavior. We recommend using a large ”known” number for the action label for such behaviors.

A.3.5 Operand attribute

The operands of instructions are declared through the **operand** attribute, with syntax:

```
operand operandname accessor ( parameters ) ;
operand – operandname ;
```

The first form specifies the name of the operand (which must have been previously declared using the **operandname** statement), the name of the accessor to use to decode, read, write, backup, or restore the operand, and the parameters to use when calling the accessor. The second form removes an operand from the instruction.

Declaring an operand does two things: it makes the operand name available for use within the instruction's actions, and it generates actions which decode and read or write the operand. This code is appended to the actions at the labels given by the original **operandname** statement.

A.3.6 Frequency attribute

The **frequency** attribute declares how frequently an instruction is used. This information is used when synthesizing the instruction decoder to improve decode performance. If the frequency is defined multiple times for an instruction, the defined frequencies are added together. The syntax is:

```
frequency expr ;
```

A.4 Sharing instruction attributes

Instruction attributes are shared through use of groups of instructions called instruction classes. Instruction classes work something like classes in object-oriented programming, though the inheritance is quite different and depends upon the attribute.

Instruction classes are defined using the following syntax:

```
instrclass classname ;
instrclass classname {
      attribute declarations
}
```

An instruction class can contain any kind of instruction attribute. Also, just as instruction definitions are ”open” and can be extended by further statements, instruction classes are ”open” and can be extended.

Instructions inherit from instruction classes through an attribute specification of the instruction with the following syntax:

```
classes classname1 , ... ;
classes –classname1 , ... ;
```

The first form adds a parent instruction class, while the second form removes a parent instruction class. Instruction classes may inherit from other instruction classes. Also, all instructions are themselves instruction classes, and may thus serve as parent classes. (Essentially, an instruction is simply an instruction class that has been marked as "a real instruction.")

The inheritance of attributes depends upon the order in which code is processed. As LIS statements are executed, the value of each attribute of each instruction and instruction class is maintained. When a parent class is added to a child class, the parent's attributes are immediately merged into the child's attributes and the parent is added to a list of parent classes for the child. When a parent class is removed from a child class, the attributes of the child are not affected, but the parent class is removed from the child's parent list. When an attribute is changed in a class which has children, the effect is as if the statement were executed on every descendant class.

There are two special instruction classes. The first is the *ALL* class, which is a parent to all other instruction classes and instructions. The second is the *DEFAULT* instruction. This instruction matches any bitfield values. By assigning behavior to this instruction, the behavior of the "illegal" opcode space can be defined.

A.5 Creating groups of instructions

There are three statements which create groups of instructions: the **cross** statement, the **instructionlist** statement, and the **instrclasslist** statement.

The **cross** statement creates a set of instructions as the cross product of instruction classes. Its syntax is:

```
cross { classname1 , ... } , ... ;
cross @ { classname1 , ... } , ... ;
```

Each list of instruction class names within curly braces is treated as a set of classes. The cross product of the sets is formed and an instruction is generated for each element of that cross product. The name of each generated instruction is formed by concatenating the name of the parent class (the class in which the statement is executed) to the names of each of the instruction classes from which the instruction was formed. Each generated instruction has as parents the parent class as well as each class from which it was formed. If any of the generated instructions already exists, it is not recreated. The second form of the statement causes all generated instructions which already exist and all of their child classes to be marked as instructions.

The **instructionlist** statement and **instrclasslist** statement define a set of instructions or instruction classes, respectively, and optionally set matches on an instruction bitfield. The syntax is:

```
instructionlist [ instructionname , ... ] ;
instrclasslist [ instructionclassname , ... ] ;
instructionlist [ instructionname , ... ] = bitfield bitrange ;
instrclasslist [ instructionclassname , ... ] = bitfield bitrange ;
```

The first two forms simply define a list of instructions or instruction classes which are to be children of the instruction (class) in which the statement is executed. The last two forms add a match to each generated instruction; the match is on the stated bitfield with a different value for each instruction; the values begin at zero and increment by one.. This is useful for defining decode tables. If there is a hole in the decoding, a '–' can be used to skip generation of an instruction at that place in the table.

If any of the classes which are to be generated already exists, it is not created anew, but match information is added as indicated by the statement. Also, if the character '@' is placed before the opening bracket, then the instruction/not-instruction status implied by the statement is propagated to any classes in the list which already exist as well as their subclasses. Use of this feature can form a convenient way to "turn off" a group of instructions.

A.6 Creating multiple levels of interface granularity

Multiple levels of interface granularity as defined in [21] are supported through the use of *buildsets*. A buildset declares *entrypoints* into the simulator, *decoders* for a set of instructions, and *shown fields*. The syntax for this is the **buildset** statement:

```
buildset buildsetname baseclass style {
  attribute declarations
}
```

This statement declares a buildset and its associated base instruction class and implementation style. The base class is used to determine the set of instructions which are to be recognized by decoders for this buildset as well as the semantics which are to be shared by all instructions before decoding occurs (e.g. fetch behavior). Implementation styles were described in 3.2. When the style is omitted, it is assumed to be unimplemented, which means that the buildset is "unimplemented" and ignored. If both the style and base class are omitted, the base class is assumed to be *ALL*.

As with the **instruction** statement, buildset declarations are open; a buildset may be defined multiple times with the attributes accumulated. Likewise, attributes may be declared outside of a **buildset** statement by inserting the buildset name immediately after the keyword.

Instruction and instruction class attributes, codesections, styles, types, operand names, accessors, instruction fields, and other buildsets can all be declared within a **buildset** statement. These declarations will only take effect if the buildset is implemented. This feature can be used to provide "libraries" of buildsets with the assurance that only the types, instruction fields, semantics, etc. that are actually needed by the implemented simulator entrypoints are generated into the simulator code.

Options and constants declared within buildsets have scope only within that buildset. The option values are seen only by decoders and entrypoints generated within the buildset.

Each of the buildset attributes will now be described.

A.6.1 Decoder attribute

The **decoder** attribute declares that a decoder should be automatically generated. The syntax of this declaration is:

```
decoder name ( parameters ) ;
decoder name ( parameters ) = action-list ;
```

The first form gives a name for the decoder and a list of extra parameters. The second form provides the name and parameters along with a list of action labels (in a format which will be described when entrypoints are explained) whose behavior is to be executed within the decoder after the instruction is decoded. The generated decoder is a function which performs a decoding of the instr instruction field based upon the match attributes of all instructions which have inherited from the base class of the buildset; the decoder returns a decode token, which is an enumerated value which is unique to the instruction. This token is used to vector to instruction-specific behavior within entrypoints. If the decoder has been given actions to perform, those actions are performed "in line" with the decoding decisions.

A.6.2 Entrypoint attribute

The **entrypoint** attribute defines an entrypoint (interface function) into the simulator. An entrypoint is a collection of instruction behavior which the user can call or which can be called from other simulator functions. An entrypoint is specified using the following syntax:

```
entrypoint return-type name ( parameters ) = action-list ;
entrypoint { return-type } name ( parameters ) = action-list ;
```

The entrypoint's signature is declared using C++ syntax, but when the return type is not a single identifier, the second form must be used.

A C++ function is generated for each entrypoint. The contents are determined by the action list. The action list consists of two lists of label expressions separated by a C++ expression evaluating to a decode

token enclosed in curly braces. Each label list is a comma-separated list of action labels which can be specified as inclusive ranges for conciseness. Thus the list 1:3,7 means "labels 1, 2, 3, and 7." The first label list specifies actions which are to be taken from the base class of the buildset. Its purpose is to specify common behavior that does not depend upon what the instruction is decoded to be, e.g. fetch and the call to the decoder. The second label list specifies actions which are taken from the individual instructions. The decode token expression is used to select the action behavior to perform from among the possible instructions. Typically the expression is simply the name of the instruction field which stores the decode token. Either the pre-expression list or the post-expression list (with the expression) may be omitted.

Instruction semantics are very broadly defined and need not be confined to normal instruction execution. For example, code to perform disassembly can be placed into actions and grouped into an entrypoint.

A.6.3 Hide and show attributes

An important element of controlling granularity is controlling the amount of information about instruction execution made available to users of the simulator. The finest element of granularity is the instruction field. Fields have a visibility property which can take two values: *shown* or *hidden*. A shown field is available to the user of the simulator; all references to the field within entrypoints and decoders refer to the field within the instruction information structure. A hidden field is not available to the user of the simulator; all references to the field within entrypoints and decoders refer to a local variable within the entrypoint or decoder.

The visibility can be controlled with the following syntax:

```
show field-name , ... ;
hide field-name , ... ;
hide & field-name , ... ;
```

The first two forms set the visibility of the listed fields to be shown or hidden, respectively. The third form sets the visibility to hidden, but also indicates that the local variable for the field should not be generated. This form is used when a field is to be replaced with a parameter to the entrypoints.

By default, all fields are hidden unless they were declared using access text. All fields which should be considered inputs to the simulator must therefore be explicitly shown as must the minimal output fields (usually "next PC" and "was there a fault".) In addition, any field which is needed to carry information between instruction steps in a particular buildset (e.g. the decode token) should be shown.