Computing System Descriptions for Systems Software

Mark W. Bailey
mark@virginia.edu

Jack W. Davidson
jwd@virginia.edu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The proliferation of high-performance microprocessors in recent years has made the development of systems software, such as compilers, assemblers, linkers, debuggers, simulators, and other related tools, more challenging than ever. When a new processor is introduced, each of these applications must be rewritten or retargeted to the new machine. This paper describes a description system, called CSDL, that permits the specification—in a concise, easily understood notation—of all aspects of a computing system that must be known in order to automate the construction of high-quality systems software. Unlike past machine description languages, and as the term computing system indicates, this new description system spans the boundary between hardware and software. CSDL descriptions are modular and extensible, providing a flexible system for specifying computing system information that can be shared among many different applications.

1 Introduction

The proliferation of high-performance microprocessors in recent years has made the development of *systems software*, such as compilers, assemblers, linkers, debuggers, simulators, and other related tools, more challenging than ever. Systems software requires detailed information about the target machine. Because this information changes from machine to machine, significant portions of systems software must be modified when a new machine is introduced. This paper describes a description system that permits the specification—in a concise, easily understood notation—of all aspects of a computing system that must be known in order to automate the construction of high-quality systems software. Unlike past machine description languages, and as the term *computing system* indicates, this new description system spans the boundary between hardware and software.

The implementation of systems software is difficult and time consuming. To remain competitive, it is essential that software design efforts produce applications that are portable so their high development costs may be amortized across a range of hardware platforms. An important aspect of portable software is retargetability. Systems software is *retargetable* if its *target machine*[1] can easily be changed. To improve retargetability, implementations are parameterized using descriptions of the target machine that can easily be modified when the implementation is moved from one hardware platform to another.

Techniques for describing machines fall into two categories: machine descriptions and computer hardware description languages (CHDL's). *Machine descriptions* describe details about the target

1. When an application models a machine as part of its operation, the machine is called the *target machine*.

machine's instruction-set architecture, while CHDL's describe the organization of computer designs. Although machine descriptions and CHDL's may appear to be similar, they differ in two important ways: their purpose and level of abstraction.

Traditionally, machine descriptions have been used with systems software [Fra77, GG78, GH84, Cat78, DF84, Ben89, BHE91, RF95], while CHDL's have been used, primarily, for simulation or synthesis of hardware designs [Coe89, Das89]. This difference in purpose is reflected in the kind of information the description language provides and the way that it is presented.

1.1 Motivation

Machines can be described at many levels of detail, as shown in Figure 3. Each level is represented as a cross-section of the machine. At any given level, the machine is viewed as a set of objects, such as transistors, registers, or instructions that correspond to objects at other layers. These levels of detail are called *abstraction levels*.

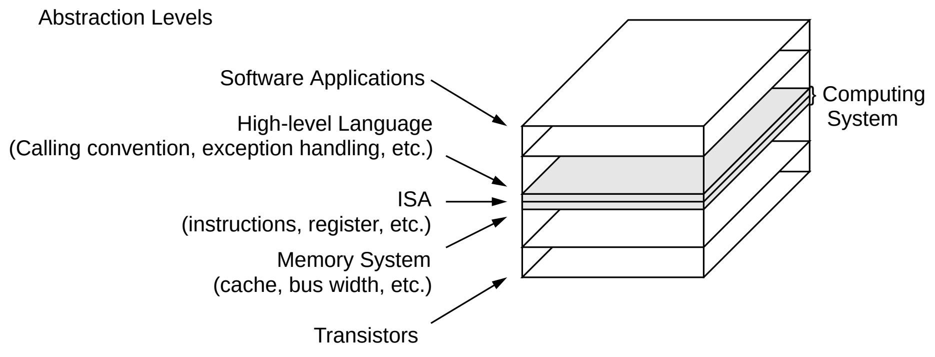


Figure 1: Abstraction levels of a machine

An important characteristic of a description technique is the level of abstraction at which the system views the target machine. The level of abstraction for a language refers to the logical level of computer design that the language most naturally describes. Examples of abstraction levels include register transfer, microprogramming and microarchitecture. Languages that are best suited for a particular design level, such as the register transfer, typically have a notion of objects native to the design level (*e.g.*, registers). The direct support of such objects in CHDL's gives them their expressive power, but limits their scope of applicability. The support of objects at a particular abstraction level makes descriptions at that level natural to read and write, while making the description at other levels, whose objects are not directly supported, awkward if even possible.

The most important decision when designing a description system is what level of abstraction will be supported. This decision is fundamental because choosing one level of abstraction excludes applications that view the target machine at a different level. Thus, the choice of what abstraction level to present implicitly selects the class of applications that can make use of the descriptions.

From the above description, it should be clear that CHDL's are quite effective at providing target-machine information for use in simulation and synthesis of *hardware* designs. However, systems software views the target machine at the computing-system level of abstraction. The computing-system level takes a wider view of the machine, as indicated by the shaded portion of Figure 3. To satisfy the requirements of applications software, we must reach slightly below the ISA (instruction-set architecture) level, and slightly above. Thus, while the ISA is essentially a cross section, the computing-system level has depth as well.

The description system we present here uses the computing-system level of abstraction in conjunction with an extensible, modular design and to promote sharing of descriptions among applications. Such sharing of descriptions that have no application bias, has many advantages, as outlined in the following section.

1.2 Application Independence

The use of a machine description can significantly reduce the retarget-time of an application. However, with each retarget of the application, a description for the new target machine must be written. For an application of any substance, this itself can be a daunting task. There are three sources of difficulty:

1. Information about the machine must be found, encoded using whatever description technique is used, and it must be tested, verified, and debugged to ensure accuracy. For some machines, finding the information is itself difficult. For some applications, the sheer volume of information to be encoded is a significant obstacle.
2. A description system that is tailored for a particular application usually contains bias toward that application. Thus, for example, a retargetable compilation system may include a machine description facility. This facility may require that information be encoded in a particular way, or that only some information be encoded. Typically, only an expert familiar with the compiler can write such a description even though the concepts that are described do not require expertise in compilers to understand.
3. Because the application does not share a common description format with other applications, one can be certain that there is not already a description available for one's use.

Using a common description format that contains no application bias eliminates these three sources of difficulties. Such a description facility is called *application independent*. Obviously for an application independent description it may at least be possible that the description already exists for the new target machine (source 3). Further, no knowledge of a particular application is required to successfully write a description (source 2). Thus any computer professional who is familiar with the machine should be qualified to write a description. Finally, if an application-independent description system becomes widely used, finding information about a target machine should become easier since computer manufacturers could supply documentation about the machine in the form of a system description (source 1).

2 A Framework for Building Machine Descriptions

This work extends previous work in machine descriptions in three key ways: abstraction level, extensibility, and modularity. By doing so, can build multi-application—if not application independent—machine descriptions. Because this new description system widens the abstraction level of machine descriptions, we call them *computing systems descriptions* to reflect their broader applicability. Our description system, called *Computing System Description Language* (CSDL), is a framework for developing more thorough, complete descriptions of target machines for use in retargetable systems software implementations.

Traditionally, machine descriptions have been monolithic entities; a single language was used to express all of the features of a target machine. Conversely, CHDL's have developed into highly modular descriptions that manage to describe large and complex hardware designs. We have chosen to exploit modularity to the fullest extent in our description framework.

As shown in Figure 4, CSDL is a framework that divides computing-system information into modules, or components. One component is distinguished from all the others: it contains the core description for the system. The *core* contains the description of the instruction set of the machine. As its name implies, it is required to be present in all CSDL descriptions, while the other components may be optionally added or removed. The description of the instruction set, which is needed in virtually all systems software, gives an otherwise amorphous system a coherent structure. Unlike the optional components, where nothing but the most minimal structure is imposed, the core's structure, or format is defined by CSDL. By defining its structure, we can insure that the most widely used component of the system is application independent, thereby promoting its adoption by many applications.

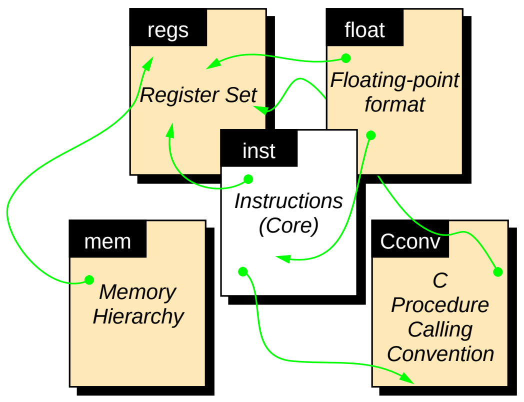


Figure 2: Computing system description framework

In addition to the core, CSDL incorporates application-defined components. A component provides additional information that is of interest to some, but not necessarily all, systems software. Since a component is application defined, it can present the information at the level of abstraction that is most appropriate for the defining application. Examples of components include pipeline and memory descriptions for different implementations of the same architecture, object file formats used by the assembler and linker, and high-level-language procedure calling conventions.

By providing modular descriptions, applications only need to examine the parts they are concerned with. Thus, descriptions need not be “complete” to be valid or useful. Different machine models might share certain parts of a description, but distinct models might have different pipeline descriptions or memory interface descriptions. Modularity also supports ease of modification. A new model of a machine might have a different pipeline, but the ISA and calling conventions likely remain the same. Only that part of the description involving the pipeline needs to be modified. Similarly, modularity helps keep the various pieces of a system description concise. The component that describes the pipeline does just that, and nothing else.

This framework has several advantages. First, it is structured enough that we can insure that the most widely shared machine feature—the instruction set—is always present in the description in an application-independent format that makes the information available to all applications. Second, the framework is flexible enough that it can provide machine specific details to a wide variety of applications. The application-defined components provide the flexibility to allow applications to define features using abstractions appropriate for the application and machine features. Third, since there frequently are duplicate applications for a particular machine (*e.g.*, two assemblers) multiple application-defined components for the same features (*e.g.*, two assembly languages) may reside in a single CSDL description. This eliminates the need for duplication within the descriptions, since they can share common information.

There is a shortcoming to this approach, however. Although we can insure application independence within the core, we cannot make any such assurances about how application dependent any of the application-defined components may be. Nevertheless, no other existing system can make this claim either. Further, we believe that there are enough benefits to presenting machine features without application bias, that, given the opportunity that CSDL provides, application developers will seek to share information between applications, and therefore strive for application independence in their description components.

3 Glue

Because CSDL descriptions are modular, significant flexibility is available to each application. The disadvantage of dividing up descriptions into smaller more manageable pieces is that this isolates each module. Without additional support, each component is likely to encounter the same pitfalls that many modular systems have: repetition among modules, and inconsistency between modules. To counter this tendency, CSDL has several mechanisms that aid in preventing inconsistency and repetition its modules. These mechanisms are the *glue* that hold CSDL descriptions together, and give them their descriptive power.

The glue imposes a minimal amount of structure on each component, in return for the following capabilities:

- import/export of values between modules,

- attachment of application-specific information, and
- support for different module facets for applications.

Each of these features is available to the member components. These features facilitate:

- modular development,
- information sharing, and
- code reuse.

In the following sections, we present each of these facilities.

3.1 Linked Values

A disadvantage of dividing descriptions into modules is that it is common for two or more modules to need access to the same information. To promote the sharing of common information between modules, CSDL provides a mechanism for introducing *linked values*.

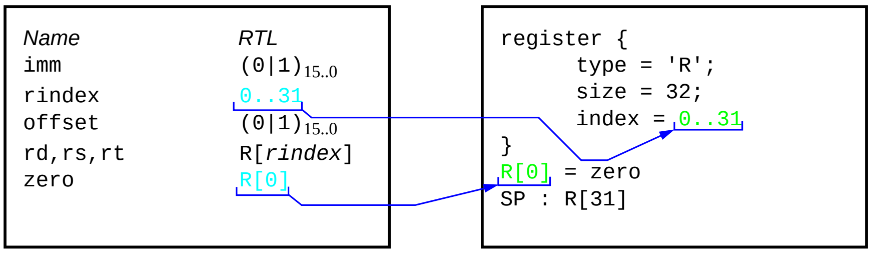
Any module may introduce a name/value pair. For example, a register description would want to be able to introduce names and values for the following registers:

- the program counter,
- the stack pointer,
- a register that is always zero, and
- the register that contains a routine’s return address.

Using CSDL’s naming system, the register description can easily provide names and values for each of these registers. These names can then be subsequently referenced in other modules. This makes it possible to easily modify the description. Although the convention about which register contains the stack pointer must be written down, it is only written down once. The value can then be propagated though the system to the other modules using links.

Figure 5 demonstrates module linking. A register description excerpt, shown in Figure 5b, defines the valid register indices as well as defining register zero (R[0]) as always storing the value zero. An instruction description excerpt, shown in Figure 5a, contains references to these two values. To accurately define the valid instructions for the machine, the instruction description must know what register indexes are valid. The instruction description refers to the valid register indices by name. Changes to the register description are immediately reflected in the each referencing module. The reference to register zero in the instruction description is discussed later.

The definition of values and their successive reference in other modules creates a web of information. These linked values are hypertext values that facilitate navigation throughout the description system. These links represent the relationship between objects in different modules. The reader of a description can better understand the interaction between objects in different description components because of the explicit representation of value references.



```
Name          RTL
imm           (0|1)15..0
rindex        0..31
offset        (0|1)15..0
rd,rs,rt      R[rindex]
zero          R[0]
```

```
register {
      type = 'R';
      size = 32;
      index = 0..31
}
R[0] = zero
SP : R[31]
```

Figure 3a: Instruction module excerpt **Figure 3b:** Register module excerpt

Figure 3: Linked values

3.2 Application Annotations

The primary shortcoming of previous machine description techniques is they present information in an application-dependent way. One source of application-specific information in machine descriptions is the natural tendency to include application-specific information about machine objects in descriptions. However, the inclusion of this information makes the descriptions application specific, and possibly useless for other purposes. CSDL provides *application annotations* to satisfy these needs and to discourage creating application specific descriptions.

Figure 6 shows the conceptual view of CSDL annotations. Annotations are pieces of information that are attached to existing descriptions for an application. Annotations are tagged as belonging to a particular application. When that application is viewing the description, the annotations appear as part of it. When other applications view the description, the annotations are not present. Annotations can be thought of as an overlay, as shown in Figure 7, that an application places over a module. The application can scribble whatever information it wishes without effecting other applications that are using the same module.

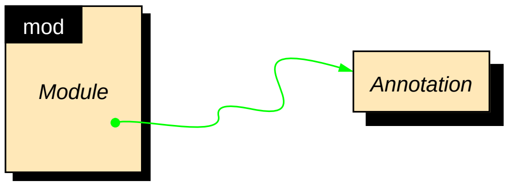


Figure 4: CSDL's annotation concept

To illustrate the use of annotations, consider a compiler that uses information in the core instruction module for generating assembly language instructions for the MIPS R2000. The compiler needs to generate an instruction to move a value from one register to another. However, the MIPS does not explicitly provide a register-to-register move instruction. The instruction description is pure[2], that is, it contains no synthetic instructions. Thus, no move instruction is listed. On the MIPS, a logical-or instruction is used, with register R[0] as the second operand, to synthesize the move instruction. If the compiler cannot glean

2. A pure description contains no synthetic or artificial instructions. We forbid the use of such impurities so that applications that depend on pure descriptions are not mislead.

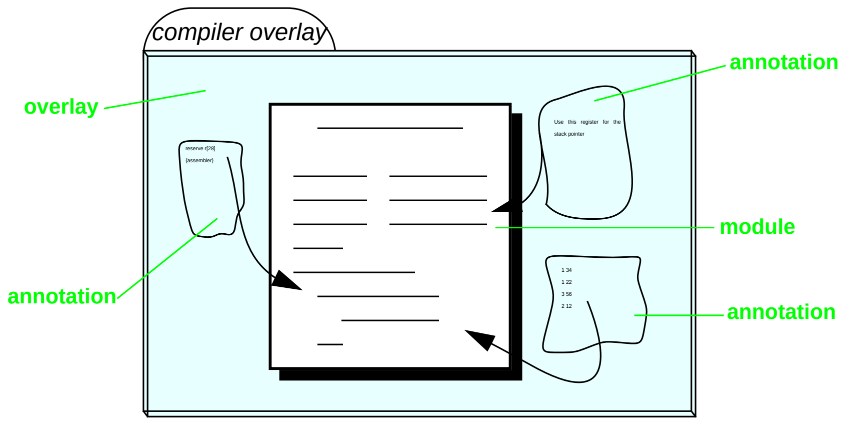


Figure 5: An application's annotation overlay

this information from the description, an annotation can be attached to the OR instruction, as shown in Figure 8, to indicate that a specific form may be used to achieve the move.

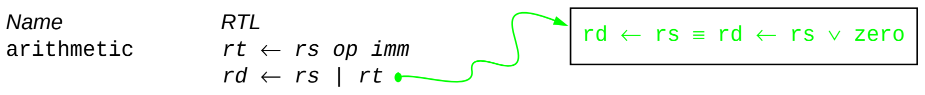


Figure 6: Annotations

3.3 Module Aspects

A concept closely related to annotations are module aspects. Annotations may be used to attach small amounts of information to selective parts of a module. For situations where more significant additions to modules are necessitated, CSDL provides *module aspects*.

A compiler's instruction description may include an enormous amount of information: semantics of the instructions, assembler mnemonics, binary format, instruction costs, pipeline scheduling information, *etc*. However, much of this information is not contained in the core description for instructions. Many applications may only have use for the semantics of the instructions and the assembler format. Each section of the description can be tagged as an aspect. An aspect is another view of an object in the description. The aspect is used to selectively filter the descriptions. Just as annotations can be viewed as overlays, aspects can as well. However, unlike an annotation overlay that is tagged for a particular application, an aspect overlay is available for use by any application. Thus, if a compiler is only interested in the semantics, instruction cost, and binary format, only those overlays are taken from the overlay "library", as shown in Figure 9, and placed over the module. This provides a mechanism for components to have many facets that are used by many applications.

Additionally, sometimes an application may wish to attach to each instruction an application-specific piece of information. Many compilers include the "cost" of an instruction. This cost may be the time it takes the instruction to execute. Another application may have a different view of the "cost" of an

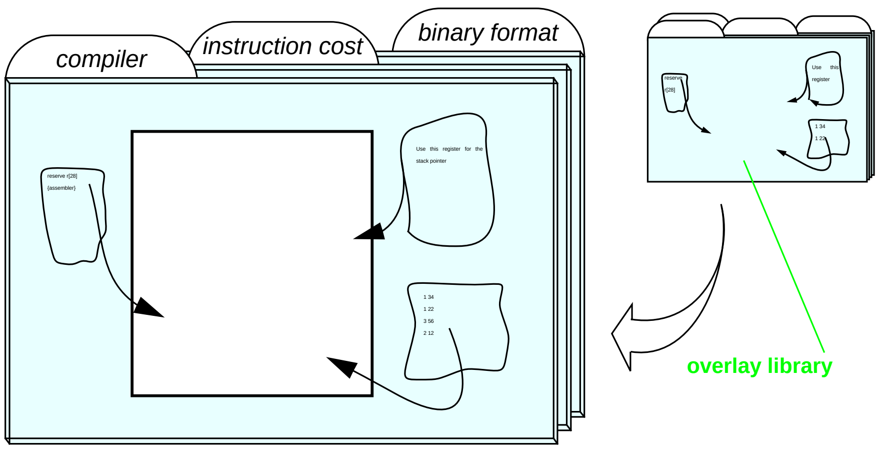


Figure 7: Constructing descriptions using overlays

instruction, such as the size of the instruction, the amount of power the instruction consumes, or the instruction’s resource demands. By providing a “cost” aspect, each instruction’s information can be augmented by the application’s concept of “cost.”

Figure 10 illustrates the use of aspects. In this case, the core description is augmented with two aspects: an assembly language aspect and a binary format aspect. For ease of identification, the assembly aspect is indicated using the magenta, while the binary format aspect appears in red. In each case, each element of an aspect has a corresponding element in the original module. So, for our example, each element of the binary format and assembly language aspects is associated with an instruction, or other object in the instruction description.

Name	*RTL*		
`rd,rs,rt`	`R[`*`rindex`*`]`	*`$rindex`*	*`rindex`*
`zero`	`R[0]`	`$0`	`0`
`op`	`+`	`addi`	`ADDI`
	$+_u$	`addiu`	`ADDIU`
`arithmetic`	*`rt`* ← *`rs op imm`*	*`op rt,rs,imm`*	`[`*`op,rs,rt,imm`*`]`
	`rd` ← *`rs op1 rt`*	*`op1 rd,rs,rt`*	`[SP,`*`rs,rt,rd`*`,0,`*`op1`*`]`

Figure 8: Assembly language and binary format aspects of instructions

Although aspects and annotations share many similar features, they differ in several ways:

1. Annotations are associated with an application, while aspects are associated with a purpose.
2. Aspects may be selectively included on a per-aspect basis, while all annotations for a given application are either included or excluded.
3. Annotations usually hold a single piece of information, while aspects contain many homogenous pieces of information.
4. Annotations are private, belonging to a single application, while aspects may be used for sharing module augmentations between applications.

4 Modules

CSDL uses modules to facilitate the extensions of descriptions by applications. In this section, we present the core module, whose format is defined by CSDL, and several other components that we have developed for applications. We chose these components because they describe computing system information that is not present in other description systems.

4.1 Core Component

The core component is the only component that is required to be present in a CSDL description. The core contains the description of the instructions for a target machine.

An instruction definition consists of two fields. The first field names the instruction (for reference in other modules), the second field gives the RTL (Register Transfer List) description of the instruction [DF80]. Although these two fields are the only required ones, additional information about each instruction may be attached using CSDL aspects. Figure 11 contains an example core instruction description for a subset of the MIPS R2000 instruction set.

Name	*RTL*
%definitions	
imm	$(0\|1)_{15..0}$
findex	
rindex	0..31
offset	$(0\|1)_{15..0}$
rd,rs,rt	R[*rindex*]
findex	31..0
fd,fs,ft	F[*findex*]
zero	R[0]
op	$+$
	$+_u$
	$\wedge$
	$\vee$
	$\oplus$
op1	*op*
	$-$
	$-_u$
	$\vee$
fop	$+_f$
	$-_f$
lop	$\leftarrow_8$
	$\leftarrow_{u,8}$
	$\leftarrow_{16}$
	$\leftarrow_{u,16}$
	$\leftarrow$
addr	*rs + offset*
%instructions	
arithmetic	$rt \leftarrow rs\ op\ imm$
	$rd \leftarrow rs\ op1\ rt$
	$fd \leftarrow_f fs\ fop\ ft$
store	M[*addr*] *sop rt*
load	*rt lop* M[*addr*]
	$rt \leftarrow imm$

Figure 9: Core Description

The first thing to notice about the description is the use of typographical conventions to facilitate the specification of the machine's instructions. This approach helps accomplish two of our primary design goals, conciseness and naturalness. Use of subscripts and italics conveys maximum information in minimum space. Second, in contrast to a simple ASCII text file, it provides a more natural way to describe machine operations.

An advantage of using typography is that a simple, concise convention indicates the portions of the description that are literals, meta-symbols, and predefined elements. Literal elements are set in the normal font, meta-symbols are set in italics, and predefined elements (*e.g.*, constants, symbols, *etc.*) are set in bold italics. These conventions allow a maximum amount of information to be conveyed in a minimum amount of space. Furthermore, it highlights the sharing of information between parts of a description. For example, to facilitate automatic syntax-directed translation, italicized symbols specify the information that must be translated from one format to another. For example, consider the specification of the store word instruction, store. To translate from RTL form to assembly language, the RTL corresponding to the meta-symbols addr sop and rt will be translated to assembly language and substituted appropriately in the assembly language template. The earlier definitions of addr, sop and rt define the translation of these symbols from RTL to assembly language.

The use of typography also facilitates the clear, natural description of machine operations. For example, consider the problems that are encountered when attempting to describe the various addition instructions a machine might have. With an ASCII file, there is only one character that naturally denotes addition, '+'. However, most machines have several types of addition they can perform. Common ones are fixed-point addition on various word sizes, and floating-point addition, again on several different formats.

A common approach to describing these different types of addition is to overload the addition operator and to determine the type of operation by determining the type of the operands. For programming languages where variables have types, this works reasonably well. However, for machines where storage cells (*i.e.*, registers) contain bits that can be interpreted differently depending on the operation applied, such a scheme does not work. In our previous version of RTL's, we solved this problem by having different names for registers depending on the type of information they held [Ben89]. So, for example, on the R2010, a floating-point register is a logical register than can hold either single- or double-precision floating-point values, so f[2] and d[2] might be used to name floating-point register two. The name f[2] indicates the register holds a single-precision value, whereas d[2] indicates the register holds a double-precision value. Thus, the RTL's

```
f[2] ← f[2] + f[4]
```

and

```
d[2] ← d[2] + d[4]
```

could be used to indicate single-precision and double-precision floating-point addition, respectively. The type of addition is derived from the type of the operands. This, however, is an unnatural view of the R2010 architecture. There are 32 floating-point general-purpose registers, and they are typically referred to as f0 through `f31`. Furthermore, single- and double-precision operations are performed on floating-point registers which are even/odd pairs of floating-point general-purpose registers.[3] It is confusing to see RTL's referencing contrived registers.

Our approach to this problem is to use typed operators. By default, '+' indicates 32-bit two's complement addition. Other types of addition must be explicitly indicated. This is illustrated in Figure 11 by the integer and single-precision add instructions.

4.2 Floating-point component

A component of CSDL not found in any other description system is our description of floating-point representations. In the past, floating-point number encodings have not been included in machine descriptions. For most applications, this information is not necessary. However, for a small number of applications, this information must be made available. Such applications include assemblers, cross compilers, and heterogenous computing applications.

CSDL is the perfect framework for incorporating such specialized, yet machine-specific information. Using CSDL, a small component can be added which describes the floating-point representation. Because this is an add-on component, the language can be defined by the application to be most descriptive of the information to be conveyed. Figure 12 shows a description of a floating-point representation. The language for describing the format is not large. However, the abstractions used are like no other abstractions in machine description languages. So, rather than exclude the possibility of describing floating-point formats at all, CSDL provides an environment including such descriptions.

Floating-point values are usually represented by a sign bit, an significand, and an exponent. A radix and radix point are not specified, but rather, implied. Despite this fairly standard method of representing floating-point numbers, numerous formats persist. The floating-point format shown here specifies the IEEE format for single-precision floating-point numbers [IEE85]. First the locations and format (sign-magnitude, twos-complement, bias, *etc.*) of the exponent and significand, and assumed radix are stated. Then, special interpretations for selected bit patterns, such as the denormal format (where the exponent is constant) for specifying extremely small numbers, NaN (Not a number), and infinity may be included.

More detailed information about this description component may be found in [BD95a]. Amongst other purposes, these descriptions may be used for:

- automatically generating conversion routines for cross compilers,
- floating-point constant folding for cross compilers,
- transmission of floating-point values between heterogenous architectures, and
- automatic generation of I/O routines for high-level language run-time libraries.

3. Kane contains a complete description of the R2010 floating-point coprocessor [KH92].

```
IEEE_single {
      exponent : bits (30-23) - 127 {
            units of bit(23) = 2^0;
      };
      significand : 1 + bits (22-0) {
            sign : bit(31);
            units of bit(22) = 2^-1;
      };
      radix = 2;
      “INF” : bits(31) = 0, bits(30-23) = 0xff, bits(22-0) = 0;
      “-INF” : bits(31) = 1, bits(30-23) = 0xff, bits(22-0) = 0;
      “NaN” : bits(30-23) = 0xff, bits(22-0) <> 0;
      zero : bits(31-0) = 0 {
            exponent = 0;
            significand = 1;
      };
      negzero : bit(31) = 1, bits(30-0) = 0 {
            exponent = 0;
            significand = -1;
      };
      denormal : bits(30-23) = 0, bits(22-0) <> 0 {
            exponent = -126;
            significand : bits(22-0) {
                  sign : bit(31);
                  units of bit(22) = 2^-1;
            };
      };
};
```

Figure 10: IEEE single precision floating-point format description

4.3 Other components

As well as the core and floating-point format components, we have developed two other components that may be included in a CSDL description: a register-set component and a component for specifying the calling convention for functions written in the C language. Additional information about the latter can be found in [BD95c].

5 Implementation

CSDL is a framework for building computing system descriptions for use by systems software. The framework itself does not mandate the format, or language of the modules that it contains. Each application that uses CSDL is responsible for the parsing and interpretation of components (component processors) that it uses. Where possible applications can share processors for components that they have in common.

CSDL components are created, edited and stored using a WYSIWYG desktop publishing system. The choice of a WYSIWYG editor for CSDL descriptions is a natural one because of the extended character sets, and fonts that many CSDL components use. Further, the glue features of CSDL (annotations, aspects, and linked values) all tag information in the description. These tags are similar to character and paragraph tags that are used in document publishing systems. Just as the formatting tags in a WYSIWYG system are hidden from the user, the tags used to maintain links and attachments are hidden when editing CSDL components.

Figure 13 illustrates the structure of the phases that we use for processing CSDL calling convention descriptions. The convention descriptions are written in a language called CCL [BD95c]. First, the CCL description is run through the CSDL description presenter. The presenter's job is to provide the view of the CSDL module that the participating application desires. In this case, the presenter overlays the calling convention component with the CCL overlay, and any other aspect overlays that the CCL front-end selects. The new convention description is then presented to the CCL front-end which interprets the CCL description and builds an intermediate representation (IR). The IR is passed to either one of several back-ends: the test case generator, the compiler table generator, or a stack walking module of a debugger. The compiler table generator uses information in the CCL description to build tables that are used to automatically generate sequences of instructions that implement the described calling convention, while the test case generator is used for generating programs that exercise a compiler's implementation of the described calling convention [BD95b].

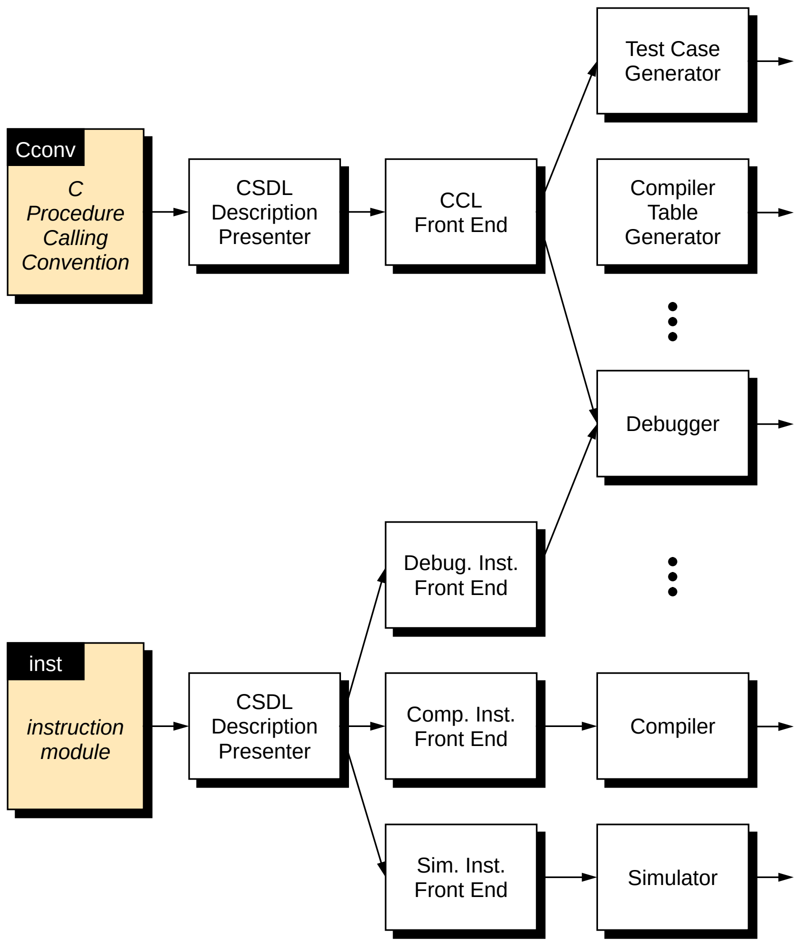


Figure 11: Processing of CSDL Descriptions

The processing of any other module in CSDL is performed in a similar manner. The CSDL description presenter is shared among all applications. In our CCL example, two different applications (a test case generator, and a compiler) make use of the calling convention description. Because the CCL description is application independent, the CCL front end can be shared among all applications that use CCL descriptions. This illustrates the power of writing application-independent descriptions.

6 Summary

In this paper, we have presented a new framework for developing descriptions of computing systems. The CSDL system presents a view of the target machine that is wider than that given by previous machine description techniques. This view—an abstraction level which spans the hardware/software boundary—is more appropriate for a variety of software applications including assemblers, linkers, debuggers, and compilers.

Systems applications need varying degrees of information about the target machine. Much of this information, especially the machine’s instruction set, can be shared among systems software. CSDL uses modules and linked values to facilitate sharing of information, and incremental development of descriptions.

However, because it is difficult, if not impossible, to anticipate all of the information that all applications will need about a target machine, there always will be a need to add information to existing descriptions. By introducing annotations, modules, and aspects, our description system makes it possible to make these necessary extensions—without impacting existing applications. When details about a target machine are missing from a description, an application can extend the description system in whatever way is most appropriate for the application’s purposes.

Finally, choosing the CSDL system for parameterizing an application does not preclude the use of an already proven system of description. Instead, with few, or no modifications, an extant description can be integrated with CSDL, enhancing its descriptive capability and making it available for other applications to use and extend.

References

[BD95a] Mark W. Bailey and Jack W. Davidson. Describing the representation of floating-point values. Technical report 95-43, Department of Computer Science, University of Virginia, Charlottesville, VA, September 1995.

[BD95b] Mark W. Bailey and Jack W. Davidson. Exhaustive testing of procedure calling sequence generation in compilers. Technical report 95-44, Department of Computer Science, University of Virginia, Charlottesville, VA, September 1995.

[BD95c] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of the 22nd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298—310, January 1995.

[Ben89] Manuel E. Benitez. A global object code optimizer. Master' s thesis, Department of Computer Science, University of Virginia, Charlottesville, VA, January 1989.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. In *Proceedings of the ACM SIGPLAN ' 91 Conference on Programming Language Design and Implementation*, pages 229—240, June 1991.

[Cat78] Roderic G. G. Cattell. Using machine descriptions for automatic derivation of code generators. In *Proceedings Third Jerusalem Conference on Information Technology*, pages 503—507, 1978.

[Coe89] David R. Coelho. *The VHDL Handbook. Kluwer Academic Publishers, 1989.*

[Das89] Subrata Dasgupta. *Computer Architecture: A Modern Synthesis, Volume 2: Advanced Topics. John Wiley and Sons, 1989.*

[DF80] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191—202, April 1980.

[DF84] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505—526, October 1984.

[Fra77] Christopher Warwick Fraser. *Automatic Generation of Code Generators. Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT, 1977.*

[GG78] Susan L. Graham and R. Steven Glanville. The use of a machine description for compiler code generation. In *Proceedings Third Jerusalem Conference on Information Technology*, pages 509—514, 1978.

[GH84] Susan L. Graham and Robert R. Henry. Machine descriptions for compiler code generation: Experience since jcit-3. In *Proceedings Ninth Jerusalem Conference on Information Technology*, pages 236—250, 1984.

[IEE85] IEEE. IEEE standard for binary floating-point arithmetic. *SIGPLAN Notices*, 22(2):9—25, February 1985.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture. Prentice Hall, Englewood Cliffs, NJ, 1992.*

[RF95] Norman Ramsey and Mary F. Fernandez. The new jersey machine-code toolkit. In *1995 Usenix Technical Conference*, pages 289—301, January 1995.