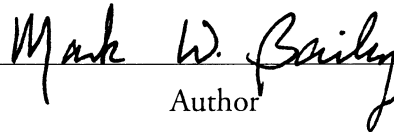
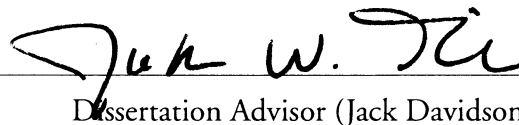


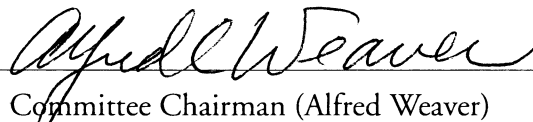
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

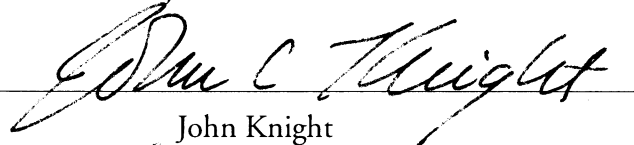

Author

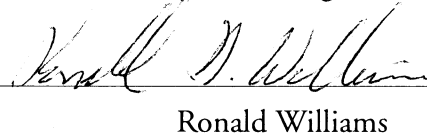
This dissertation has been read and approved by the Examining Committee:


Dissertation Advisor (Jack Davidson)

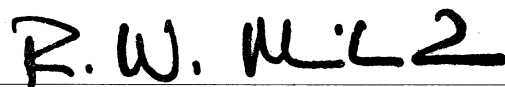

Committee Chairman (Alfred Weaver)


James Cohoon


John Knight


Ronald Williams

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

May 2000

**CSDL: REUSABLE
COMPUTING SYSTEM DESCRIPTIONS
FOR
RETARGETABLE SYSTEMS SOFTWARE**

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Mark W. Bailey

ABSTRACT

In an era of rapid design of microprocessors for desktop systems, embedded systems, and handheld computing devices, the timely construction of systems software is essential. Systems software, such as assemblers, compilers, and debuggers, must be constructed before development of application software for a microprocessor can commence. However, the implementation of such machine-specific applications is difficult and time consuming. Therefore, to remain competitive, it is imperative that systems software designs focus on portability to reduce implementation time and ensure rapid delivery of complete systems to the market. This dissertation presents the Computing System Description Language (CSDL) framework that addresses these rapid development requirements.

We illustrate the CSDL framework by developing an instruction-set description component (τ RTL), an optional procedure calling convention description component (CCL), and the mechanism we use to extend extant descriptions (CSDL). τ RTL and its accompanying microinstruction descriptions (μ RTL) further the state-of-the-art in specifying semantics of machine instructions. τ RTL adds a new type system and abstract syntax that facilitates more accurate specification and automatic detection of errors by τ RTL manipulators. τ RTL machine descriptions are also application independent—they completely separate the specification of semantics from the application's implementation. The CCL specification language is the first work to formally describe procedure calling conventions. We demonstrate two distinct uses for CCL descriptions: code generation and fault detection. Using CCL we have built compilers that are more robust, and found and diagnosed faults in production compilers. CCL, τ RTL, and μ RTL descriptions are bound together using CSDL. CSDL is the first description system to recognize that specifications must evolve and that specifications will frequently include application-dependent features. The CSDL environment provides facilities

for adding new components, sharing information between components, and extending existing components for use in a wide variety of applications.

ACKNOWLEDGMENTS

I would like to thank everyone who has contributed, supported, or encouraged me in this work.

First and foremost, I would like to thank my advisor, Jack Davidson. Over the many years, Jack has been much more than just an advisor; he's been a teacher and friend. I owe all of the good ideas in this dissertation to Jack. The bad ones are all mine.

The members of my examining committee, Jim Cohoon, Jack Davidson, John Knight, Alf Weaver, and Ron Williams had the unfortunate job of reading this dissertation cover-to-cover. I thank them for holding me to their high standards.

I could not have completed this degree without the wonderful support of the faculty, staff, and students in the Computer Science Department at the University of Virginia. There are, unfortunately, too many to list here. I've made so many friends and learned so much. In return, I will always be grateful.

All of my family supported me with encouragement and enthusiasm. In particular, my parents, Duane and Leeta, and my wife's parents, John and Carol provided much needed spiritual support. They never asked "how much longer?" at the wrong time.

Finally, I wish to thank my wife, Ann, for her love, continuous support and encouragement. I will never understand how she stuck with me during the hard times.

To Dad

CONTENTS

Chapter 1 — Introduction	1
1.1 Background	2
1.2 The Problem	4
1.3 Motivation	5
1.4 Structure	6
Chapter 2 — Computing System Descriptions	8
2.1 Computer Hardware Description Languages	8
2.1.1 VHDL	9
2.1.2 ISP	11
2.1.3 Lisas	12
2.2 Machine Descriptions	13
2.2.1 ISP'	13
2.2.2 TMDL	13
2.2.3 MDL	15
2.2.4 Mop	15
2.2.5 PO and VPO	16
2.2.6 The GNU C Compiler	18
2.2.7 Maril	19
2.3 Multipurpose Descriptions	20
2.3.1 SLED	20
2.3.2 λ -RTL	20
2.4 Summary	21
Chapter 3 — Specifying Instruction Semantics: CSDL Core Descriptions	22
3.1 String RTL's	23
3.1.1 String RTL Syntax and Semantics	23
3.1.2 Analysis and Manipulation	28
3.2 τ RTL's	31

3.2.1	Syntax	31
3.2.2	τ RTL Types	34
3.2.3	Aliasing	36
3.2.4	Notation	38
3.2.5	Abstract Syntax	39
3.3	Using τ RTL's to Describe Machines	44
3.4	Operation Semantics – μ RTL's	47
3.5	Summary	52
Chapter 4	— Specifying Procedure Calling Conventions	54
4.1	Introduction	54
4.1.1	Motivation	55
4.1.2	Applications	56
4.2	Procedure Calling Conventions	57
4.2.1	A Simple Calling Convention	57
4.2.2	Convention, Language, and Implementation	58
4.2.3	Separating Convention from Sequence	59
4.2.4	Interfaces and Agents	60
4.2.5	Addressing	61
4.2.6	Activation Frame Layout	62
4.3	The CCL Specification Language	62
4.3.1	Design Philosophy	62
4.3.2	Resources	63
4.3.3	Global Section	64
4.3.4	Agent Descriptions	65
4.3.5	Summary	70
4.4	The Formal Model	70
4.4.1	P-FSA Representation	70
4.4.2	Automatic P-FSA Construction	75
4.4.3	Completeness and Consistency in P-FSA's	77
4.5	Use in a Compiler	80
4.5.1	The Interpreter	80
4.5.2	Realizing the Calling Sequence	81
4.6	Construction of Diagnostic Programs	85
4.6.1	Test Vector Selection	85
4.6.2	Test Case Generation	90
4.6.3	Automatic Diagnosis of Errors	92
4.6.4	Test Results	95
4.7	Summary	98

Chapter 5 — Computing System Description Language	100
5.1 CSDL Overview	101
5.1.1 Modules	102
5.1.2 Linked Values	103
5.1.3 Application Annotations	104
5.1.4 Module Aspects	106
5.2 Module Processing	107
5.2.1 CSDL Language Processing	107
5.2.2 An Environment for CSDL	112
5.2.3 Processing Summary	117
5.3 Applications	118
5.3.1 Binary Translation	118
5.3.2 Specifying a Procedural Interface to Assembly Language	120
5.4 Summary	121
Chapter 6 — Conclusions	123
Appendix A — CSDL Descriptions	128
A.1 The MIPS Core Description	128
A.2 The Motorola M68020 Core Description	133
Appendix B — CCL Descriptions	137
2.1 The MIPS R3000 CCL Description	137
2.2 The M68020 CCL Description	139
2.3 The M88100 CCL Description	140
2.4 The DEC VAX-11 CCL Description	141
2.5 The SPARC CCL Description	142
References	144

LIST OF FIGURES

Chapter 1 — Introduction

Figure 1-1. Procedural machines description use	3
Figure 1-2. Declarative machine description use.	3

Chapter 2 — Computing System Descriptions

Chapter 3 — Specifying Instruction Semantics: CSDL Core Descriptions

Figure 3-1. Context-free grammar for τ RTL's	34
Figure 3-2. Memory aliases created by overlapping memory references.	37
Figure 3-3. Improperly typed τ RTL for a load	42
Figure 3-4. A properly typed τ RTL for a load.	42
Figure 3-5. Abstract syntax for two τ RTL's	43
Figure 3-6. Combined subexpression	43
Figure 3-7. Incorrect simplification of $(r[2_{u,5}]_{u,32} + \Delta(20_{u,9})_{u,32})_{u,32}$ tree	43
Figure 3-8. Correct simplification of $(r[2_{u,5}]_{u,32} + \Delta(20_{u,9})_{u,32})_{u,32}$ tree	44
Figure 3-9. An τ RTL grammar for a very simple machine	45
Figure 3-10. An illegal τ RTL grammar.	45
Figure 3-11. A properly formed τ RTL grammar.	46
Figure 3-12. A complete τ RTL machine description of the DLX	48
Figure 3-13. μ RTL operational semantics for a user-defined string copy operator	51
Figure 3-14. Operational semantics for the Pentium PADDB instruction	52

Chapter 4 — Specifying Procedure Calling Conventions

Figure 4-1. How CCL specifications are used.	56
Figure 4-2. Rules for a simple calling convention	57
Figure 4-3. The role of agents in procedure call and return interfaces.	61
Figure 4-4. The caller prologue	68
Figure 4-5. A CCL description of the calling convention of Figure 4-2	71
Figure 4-6. P-FSA for transmission of parameters for a simple calling convention	72
Figure 4-7. Algorithm to build a P-FSA	76

Figure 4-8. Definition of State-Label	77
Figure 4-9. Calling sequence locations	82
Figure 4-10. A possible procedure activation frame structure	83
Figure 4-11. Example FSA where a fault will not be detected.	87
Figure 4-12. Entering and exiting transitions for a state	88
Figure 4-13. Test vector generation algorithm	89
Figure 4-14. The compiler conformance test process	91
Figure 4-15. An example outcome	92
Figure 4-16. Determining conformance of n compilers	96

Chapter 5 — Computing System Description Language

Figure 5-1. Computing system description framework	102
Figure 5-2. Linked values	104
Figure 5-3. An application's annotation overlay	105
Figure 5-4. A CSDL annotation	106
Figure 5-5. Assembly language and binary format aspects of instructions	107
Figure 5-6. CSDL Language Dispatching	108
Figure 5-7. CSDL Grammar	108
Figure 5-8. Processing of a CSDL module	110
Figure 5-9. Specifying binary translation using a CSDL aspect	119
Figure 5-10. A small MIPS excerpt with SLED aspects	121

Chapter 6 — Conclusions

Appendix A — CSDL Descriptions

Appendix B — CCL Descriptions

LIST OF TABLES

Chapter 1 — Introduction

Chapter 2 — Computing System Descriptions

Table 2-1. Abstraction Levels and CHDL Examples (Table 3.1 in [Das89]). . .	10
---	----

Chapter 3 — Specifying Instruction Semantics: CSDL Core Descriptions

Table 3-1. Sample RTL address expressions (excerped from [Ben94])	26
Table 3-2. Summary of formats for string RTL expressions.	27
Table 3-3. Built-in RTL operator summary	40
Table 3-4. Summary of τ RTL built-in operations	50

Chapter 4 — Specifying Procedure Calling Conventions

Table 4-1. Definition of λ for example P-FSA.	75
Table 4-2. Determining agent actions from placement information	84
Table 4-3. P-FSA profiles for several calling conventions.	86
Table 4-4. Sizes of test suites for various selection methods	88
Table 4-5. All outcome configurations	94
Table 4-6. Results of running the MIPS test suite on several compilers	96

Chapter 5 — Computing System Description Language

Chapter 6 — Conclusions

Appendix A — CSDL Descriptions

Appendix B — CCL Descriptions

CHAPTER 1

INTRODUCTION

In an era of rapid design of microprocessors for desktop systems, embedded systems, and handheld computing devices, the timely construction of systems software is essential. Systems software, such as assemblers, compilers, and debuggers, must be constructed before development of application software for a microprocessor can commence. However, the implementation of such machine-specific applications is difficult and time consuming. Therefore, to remain competitive, it is imperative that systems software designs focus on portability to reduce implementation time and ensure rapid delivery of complete systems to the market.

A proven technique for building portable systems software—particularly compilers—is to isolate machine-specific details of an implementation through the use of a machine description. A *machine description* is the specification of a machine's features that the implementation needs to perform its task. The machine description is used to automatically generate the machine-specific portion of the application's implementation. In theory, the machine description focuses on describing the machine rather than describing the implementation. In practice, machine descriptions often describe not only the machine, but also the process by which the machine's features are used in the implementation. Such descriptions contain application dependencies that preclude their reuse in other applications. Unfortunately, description systems, and the machine descriptions they contain are, themselves, difficult and time consuming to construct. This research concentrates on the design of description languages that promote writing reusable computing system descriptions.

1.1 Background

A survey of machine description techniques reveals two approaches to describing machines: the procedural approach and the declarative approach. The procedural approach uses an implementation to present the features of the target machine to the application. By interpreting the implementation, the desired machine-specific features are recognized. The second approach uses a table, or database, of information. Aspects of an application's implementation that are machine-specific are located in the table for convenient access by the application.

In the procedural approach, the machine description is read by a description processor which passes through source code taken from the machine description and, optionally, generates additional code from other parts of the description. Figure 1-1 depicts this process. This method has two advantages. Foremost, it is easy to implement. Often, the description is written in a special-purpose language that is augmented by the application's implementation language. Shortcomings in the special-purpose language can easily be addressed by using the application's implementation language. The other advantage to this approach is that the description language is extensible. If the application's implementation language is used, then all of the procedural and data abstraction facilities of the implementation language are available. This approach, however, also has its limitations. First, the machine descriptions are specific to an application. This makes them difficult to reuse, even though the information they contain could be useful to other applications. Second, while the characteristics of a target machine may be easy to understand, it is usually difficult for someone unfamiliar with the application's implementation to write a description of a new target machine. Third, since the descriptions are implementations, they suffer the problems of any implementation: they are difficult to read and maintain. Fourth, since these implementations are typically written in an *ad-hoc* manner, it is difficult to prove anything about the resulting descriptions.

In the declarative approach the machine description contains little or no source code. Instead, machine-independent source code that accesses the machine-dependent table is included in the application. The description processor then produces a table in the form of code that will be accessed by the supplied access routines. This process is shown in Figure 1-2. The declarative method is significantly better than the procedural technique. First, the descriptions have a fixed format. This makes them more straightforward. The details of the target machine are simply placed in the appropriate entries in the table. Thus, the descriptions

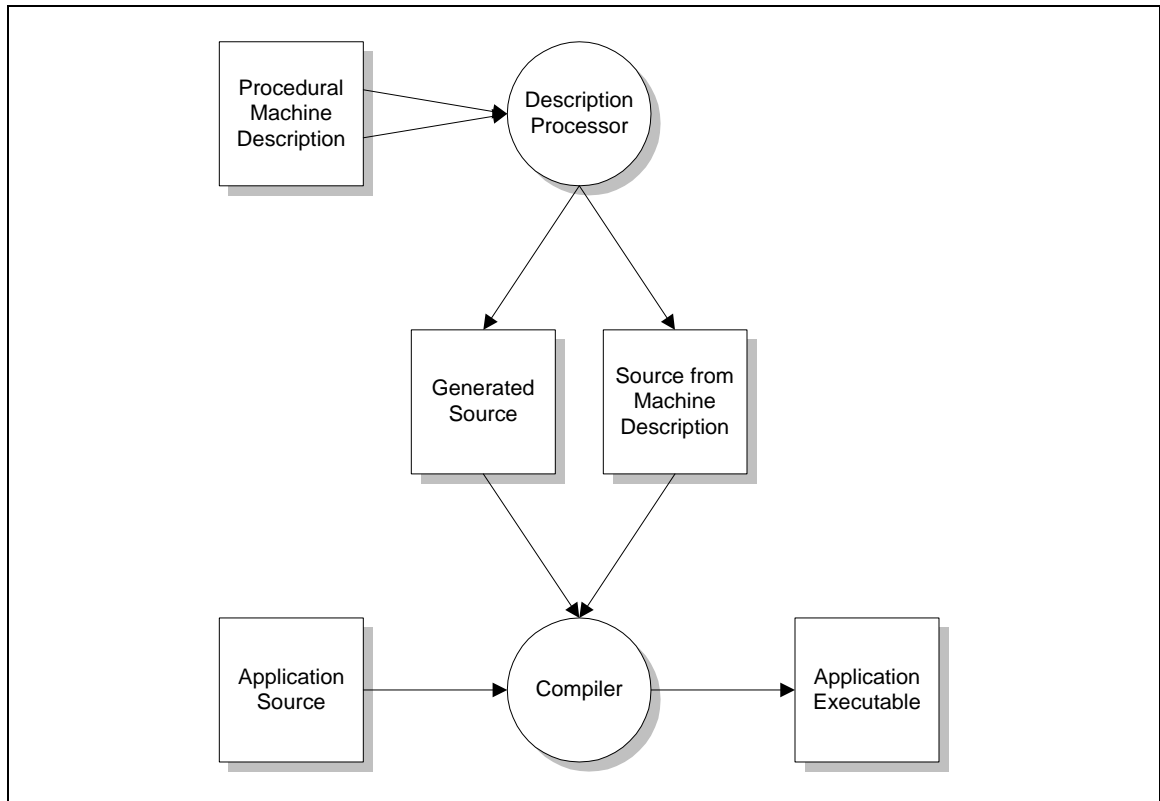


Figure 1-1. Procedural machines description use.

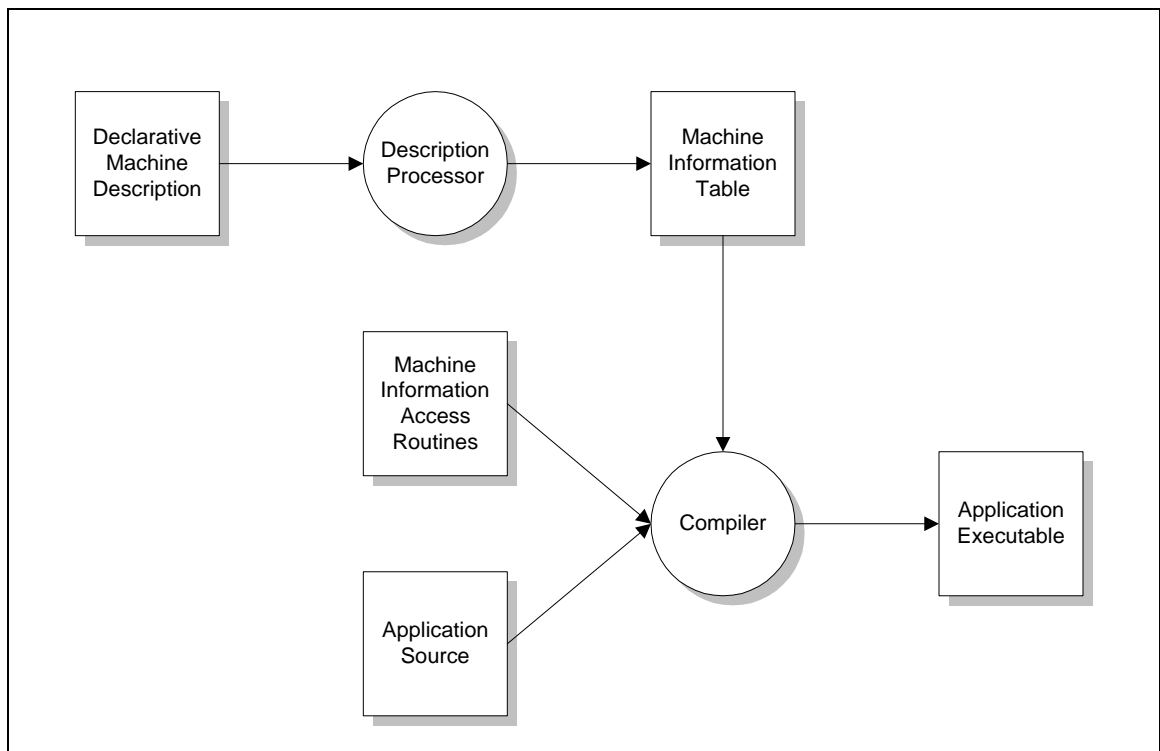


Figure 1-2. Declarative machine description use.

are more compact and concise. Consequently, they are easier to write, read and maintain. Second, a program can automatically generate, from the description tables, an implementation that is similar to the procedural approach in function and form. Also, the resulting implementation is likely to be more robust. Third, all descriptions have a similar form; similar characteristics are described in similar ways. Hence, applications using these descriptions are more easily retargeted. Fourth, this approach can be more application-independent. This approach has one disadvantage: it may be difficult to describe all of the relevant information about the target machine if the table is not general enough.

1.2 The Problem

The use of a machine description can significantly reduce the time to retarget 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, the finding of 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 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 pro-

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).

The goal of this research is to develop a more effective method for describing target machines. For the method to be *effective*, it should be application independent. The class of machines we support is the traditional von Neumann architectures. The level of abstraction is the view that applications, such as assemblers, compilers and debuggers, have of the target machines.

1.3 Motivation

Although there have been numerous efforts to design machine description techniques, many of which have been successful, none of these solutions have been very general, complete, or application-independent.

For many years, compilers have used machine descriptions to capture details about the compiler's target machine. Through the use of a machine description, target-specific information can be isolated from the rest of the implementation so that it may easily be examined and changed. Despite their success in compilers, machine descriptions have not been widely used by other systems software such as linkers, debuggers, profilers, and simulators. For the most part, where machine descriptions have been used, new systems have been developed rather borrowing the technology from an extant description system. A primary motive for this action is that machine descriptions have been *application dependent*. That is, inherent in the way the description is written is the purpose for which the application will use the information. This application dependence stifles the reuse of descriptions in other applications.

By providing a more complete description method, we can reduce the retarget time of applications. Current techniques manage only to describe a subset of the characteristics of the target machine. In doing so, these methods require that the remaining characteristics be provided in a less retargetable form.

In addition to completeness, we see the need for a more general solution. Many of the existing description systems only allow the description of a small class of machines—such as

Reduced Instruction-Set Computers (RISC's). These systems, therefore, are of limited use to retargetable applications.

Current methods of description have been designed with a specific application in mind, despite the fact that the following applications can use information about the target machine at the same level of abstraction:

- **Assemblers** - Assemblers require information about the instructions and data types of the target machine. They also require the binary format of each of the instructions.
- **Compilers** - Compilers need instruction information (both binary and symbolic), resource information (registers, functional units, busses, *etc.*), details of the subprogram calling convention, *etc.*
- **Debuggers** - For disassembly purposes, debuggers need information about the binary format of instructions and their respective symbolic form.
- **Emulators/Simulators** - These applications require information at the appropriate level of abstraction. In this case, what instructions are available and their syntax and semantics.
- **Synthesis tools** - Tools for synthesis require similar information as simulators.
- **Evaluation tools** - For example, profilers require resource and instruction information.
- **Testing tools** - Automatic testing and verification tools can use the instructions and resources as a basis for their testing.
- **Documentation** - People could use a formal description as a form of machine documentation.

Thus, there is clearly a need for an application-independent description technique. With such a description facility, all of the above tools could use a single description. This, in fact, changes the role of the description language to that of a definition language. By standardizing the descriptions, we can establish a formal method of communication among computer architects and software developers.

1.4 Structure

The following chapter presents a brief overview of previous machine description systems. Chapter three presents the CSDL (Computing System Description Language) core language used to describe a machine's instruction set. Chapter four discusses the Calling Convention

Language that we use to describe a machine's procedure calling convention. Chapter five presents the general CSDL framework that delivers flexibility and extensibility to the applications that use CSDL. Chapter six concludes by summarizing the research results and contributions of this work.

CHAPTER 2

COMPUTING SYSTEM DESCRIPTIONS

Since the 1960's researchers have investigated methods for effectively describing computing systems. Over the years, three categories of descriptions have emerged: computer hardware description languages, machine descriptions, and multipurpose descriptions. *Computer hardware description languages* (CHDL's) focus exclusively on the hardware for the purpose of simulation and synthesis of the hardware. *Machine descriptions* aim to isolate machine-specific characteristics of an implementation—typically a compiler—with the goal of making the implementation retargetable. Multipurpose descriptions aim to provide the same service as machine descriptions with the primary goal of serving a wider application audience. In this chapter, we present languages from each of these three categories in turn.

2.1 Computer Hardware Description Languages

Hardware designers started developing and using languages for the description of computer hardware systems in the 1960's. These languages, called Computer Hardware Description Languages (CHDL's) represent the earliest attempts to describe machines.

An important characteristic of a CHDL is the level of abstraction that the language was intended to be used for. The level of abstraction 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 give them their expressive power, and also limit their scope of applicability. The support of objects at a partic-

ular 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 plethora of CHDLs makes a thorough discussion of them here infeasible (Dasgupta presents a more in-depth discussion [Das89]). Table 2-1 (from [Das89]) presents examples of CHDLs representative of each of a number of levels of abstraction. Notice that a large number of these languages attempt, to some degree, to be multi-level. Since applications often view machines at a higher level of abstraction than most CHDLs are designed to present, the multi-level CHDLs appear to be the most promising candidates for building retargetable applications.

The abundance of CHDLs has given designers a large selection of description methods; it has also stifled language standardization. In an effort to alleviate the situation, the U.S. Department of Defense (DoD) has developed, as part of its Very High Speed Integrated Circuits (VHISC) project, a DoD standard CHDL called VHDL. As a result, VHDL is rapidly being adopted as an industry-wide standard CHDL. We will, therefore, review VHDL [Coe89, LSU89] which is representative of these multi-level languages.

2.1.1 VHDL

A VHDL description is composed of *design entities* that are organized hierarchically. An entity, in turn, is composed of an *interface* and one or more *bodies*. An interface defines *ports* which are the only method of communication between an entity and other entities. There are two types of bodies: structural and behavioral. A structural body simply connects the entity's ports to ports of sub-entities contained in the entity body. A behavioral body, on the other hand, specifies the behavior of an entity using a procedural language. Behavioral bodies are used to define simple entities, while structural bodies are used to hierarchically build new composite entities from existing ones.

Data objects in VHDL may be one of constant, variable, or signal. Constants and variables are similar to their counterparts in programming languages. Signals, however, are new. A *signal* is connected to an interface port, and holds a value just as a variable does, but has an additional dimension—time. Signals are changed using a signal assignment. The assignment occurs when a value in the assignment (another signal) changes value. A time

Levels of Abstraction	Examples of Languages
Architectural Exo-architecture Endo-architecture Micro-architecture	
Microprogramming Machine-independent Machine-dependent	
Register Transfer Logic design	

Table 2-1. Abstraction Levels and CHDL Examples (Table 3.1 in [Das89]).

delay may also be added to delay when the assignment takes place. Thus, signals may easily be used to model the wire connections of a computer.

In summary, VHDL uses entities to model the components of a system. Just as components are made up of sub-components, entities may be constructed using sub-entities. Wires connecting components are modeled using signals connecting ports. Finally, the behavior of the simple entities is described using a procedural language.

VHDL has a number of strengths. The hierarchical design makes it possible to manage descriptions of large, complex systems. Information about the behavior of components of the system can be precisely defined. Further, VHDL already has an established user base in simulation, design and synthesis, which could facilitate the sharing of descriptions. Unfortunately, for our purposes, VHDL's shortcomings are severe. The descriptions provide information at an inappropriate level of abstraction, making it difficult to extract the needed information.

2.1.2 ISP

The first language to deviate from describing purely hardware is Bell and Newell's ISP (Instruction-set Processor) descriptive system [BN71]. However, we still place ISP in the CHDL category. ISP focuses on characteristics of the instruction-set architecture (ISA). The purpose of the notation is to uniformly describe instruction sets of a variety of machines. An ISP description has two parts: "the nature of the operations and the rules of interpretation." As such, Bell and Newell argue that this completely describes the behavior of the machine.

A typical ISP description is divided into five sections: the processor state, instruction format, effective address calculation, instruction interpretive process, and instruction set. The processor state and instruction format sections define the names and sizes of storage locations and instruction fields, respectively. The remaining three sections use a more procedural approach. Rather than describing what an instruction does, or what addressing modes are available, ISP descriptions describe how each of these work. Addressing modes are defined in terms of operations on the previously declared storage locations. Instructions are defined by their effect on the state of the machine using a register transfer notation to indicate the semantics. Finally, the instruction interpreter is defined in a similar way by using register transfers to describe the interpreter's effect on the state of the machine.

ISP has a couple of good points. First, it is general. This is illustrated by the *forty* machine descriptions provided in Bell and Newell's book. Second, the entire "programmer's view" of the system, as defined by the programmer's manual, can be described. Third, the language provides detailed information about the binary format of the machine's instructions. This is very useful information for applications that manipulate machine code.

Bell and Newell's system has several serious disadvantages, though. First, ISP describes machines at the wrong level of abstraction. Graham notes that "ISP contained **too much** detail, making it hard to extract the needed information from the description" [GH84]. Second, the descriptive system provides no information about software conventions, which are of interest to our applications. Third, ISP is not formal; its syntax and semantics are open-ended which makes it unusable by an automated system [Lun83]. This is primarily because ISP was designed as a notation for communicating machine characteristics between people [Wic75]. Thus, for an automated system to use ISP, a number of restrictions would have to be imposed on the language.

2.1.3 LISAS

Cook and Harcourt also describe the instruction-set architecture using a specification language called LISAS [Coo94, CH94a, CH94b]. LISAS is described as a functional language that models machines as a machine state and transformations on that state. The descriptions include storage bases, access classes (instead of operand addressing), data type descriptions, and instruction formats. Cook aims for application independence and raising the level of abstraction above CHDLs.

Unlike the other description systems, LISAS was designed for instruction-set simulation. This places them squarely in the class of CHDLs. A LISAS description presents information at a level of abstraction somewhere between CHDLs and machine descriptions. Although Cook claims that LISAS can be used for applications other than simulation, it is not at all apparent how applications that generate assembly language could make use of the descriptions since they detail the binary format of instructions, but not the symbolic assembly format. LISAS primary abstraction seems to be the instruction. If one wishes to describe other architectural features, such as the instruction execution pipeline, it is not clear how one could accomplish this within the current LISAS framework.

2.2 Machine Descriptions

Unlike traditional CHDLs, Bell and Newell's ISP appealed to systems software developers. Shortly after ISP's introduction, machine descriptions emerged to aid in the construction of both assemblers and compilers. Machine descriptions are used to isolate and describe features of computing systems for retargetable software. In this section, present the most successful machine description systems.

2.2.1 ISP'

Despite its shortcomings, ISP forms a foundation for many subsequent description systems. Since ISP has never been formally defined, a number of interpretations have evolved. One such interpretation is Wick's ISP' which is used in his assembler generating system [Wic75], and also in Fraser's automatic code-generator generator [Fra77a, Fra77b]. ISP' has a formal definition for its syntax and semantics, thus enabling it to be parsed, and used by such systems. Wick's system, however, places very few demands on the machine description system. In particular, the assembler generator has no need for a description of the semantics of each instruction, although they are present. Only details such as the binary format of the instructions, their mnemonics, and the data type encodings are used [Fra77b].

2.2.2 TMDL

One of the first to abandon the ISP notation were Graham and Glanville. They use a machine description to enhance the retargetability of their table-driven code generation system [GG78b, GG78a]. Their language, called TMDL (Target Machine Description Language), uses attribute-grammar productions as its form of machine description.¹ A machine description is composed of sections that describe the resources of the machine (such as the register set) and the instruction set.

The resource description is rather limited; it allows for specification of "logical groupings of register classes and pairs," and of which registers are available for allocation. The instruction-set section, however, is much more flexible. Instructions are described using a syntax-directed translation [ASU86]. Each target machine instruction is "described" using a

1. Ganapathi and Fischer have subsequently used this technique in their description-driven code-generation system [GF82].

semantically equivalent intermediate representation (IR) expression and a template for the corresponding assembly language instruction. Code is generated for the target machine by finding an instruction in the table that matches the IR expression. The assembly language template provides the translation from the IR to the target machine's assembly language. So, TMDL is not a machine description, but instead a code generator description.

Early versions of TMDL required a separate rule for each combination of instruction and addressing mode. Graham and Henry refer to this structure as a “flat” grammar since every grammar rule corresponds to a single instruction. Thus, an instruction with three operands, each with four possible addressing modes would require $4^3 = 64$ different rules! Complete descriptions of a machine like the VAX-11 [Dig78] would be impractically large, since it would require several million grammar rules [GH84]. Later versions “factored” the grammar allowing descriptions of common portions of instructions, such as addressing modes, to be centralized [GHS82].

TMDL, in its final form, is a significant improvement over previous languages. Since the descriptions are essentially syntax-directed translations, they are easy for the implementor (in this case, a compiler writer) to understand. The original goal of isolating the machine's instruction set and assembly language format has been accomplished. Consequently, a number of machines have been described, thus providing working compilers. Finally, this approach has, to some extent, managed to separate the implementation *using* the descriptions from the descriptions themselves. This feature diverges from the previous descriptions which are more procedural—making it more suitable for other applications.

TMDL has several shortcomings, however. Although the compiler implementation has been separated from the description, the description still reflects the purpose of the implementation. The choice of the intermediate language as a method for describing the semantics of instructions reduces the usefulness of TMDL as a description system for other applications. The language requires an understanding of the IR for the compiler system and of syntax-directed translation which are skills that should not be required to describe the characteristics of machines.

Eventually, Graham and Henry abandoned TMDL altogether. A new LISP-like description language, called LISPMD (LISP machine description), was created [AGH+84]. Although LISPMD's design evolved from TMDL, its syntax and semantics diverge from it.

LISPM is much more a general pattern-processor than a description system. A description is composed of “meta-family patterns,” “meta-rules,” semantic actions, and cost and weighting factors for each instruction. The semantics and rules for macro expansion make writing, or even reading, machine descriptions daunting for someone unfamiliar with the implementation of the compiler, the description system, and LISP.

2.2.3 MDL

Boulton and Goguen developed a machine description language (MDL) to aid in the development of retargetable compilers [BG79]. In particular, MDL was designed to describe instruction-sets and memory structures in a form that could be processed by a machine. MDL is a direct descendent of ISP. Consequently, they decompose their description into structures similar to ISP's. MDL has separate facilities for describing the instruction semantics, instruction format, the structure of memory and data, and basic units (the basic addressable unit, number base, and instruction alignment). MDL is also hierarchical; basic units such as memory structure are used to describe the instruction format, which is subsequently used in the instruction description.

MDL provides a great deal of information at the bottom of its hierarchical structure. Details such as the base of the number system used by the machine and the data representation encoding (two's-complement, sign-magnitude, EBCDIC, *etc.*) are easily expressed. In addition, structures with similar properties can be grouped together, resulting in a more compact description. However, higher in the hierarchical description, where the language more closely interfaces with the intermediate language, details become more *ad-hoc*. In particular, unlike TMDL, the addressing modes for instructions are not separated from the form of the instructions. There is also no uniform model or formal language for the semantics of the instructions. Moreover, the addressing mode is implicitly derived from the format of the instructions. This greatly restricts the variety of instructions that can be described by the model.

2.2.4 Mop

Cattell designed an instruction-set formalism for use in a machine-independent code generator for the PQCC (Production-Quality Compiler-Compiler) project at Carnegie Mellon University [LCH+80]. Cattell uses a declarative, rather than a procedural description called Mop

[Cat78, Cat80]. The model assumes, as ISP does, a machine composed of a processor and memory. Information provided by the model is divided into five categories:

1. *Storage bases*—locations that store the processor state. Each location is assigned a type, such as primary memory, reserved, or temporary.
2. *Operand addressing*—defined using an expression in terms of storage bases.
3. *Machine operations*—semantics of each instruction in terms of input and output assertions on the processor state. The semantics are described using a tree notation similar to the TMDLs. Attached to each instruction is its cost.
4. *Data Types*—size, type, and encoding of each supported data type.
5. *Instruction fields and formats*—the format and encoding of each of the machine instructions described in the machine operations section. These also include the type and operand class of each instruction field.

As mentioned earlier, Mop is used with a machine-independent code generator. Cattell identifies problems in the interface between the description and the code generator. Specifically, he discovered that a set of general axioms were required to transform some intermediate language forms into different, equivalent forms that would match the semantic descriptions of the machine operations. The axioms are used to express the identity and commutativity relations for the operators in the intermediate language.

Many of the problems with TMDL descriptions can be found in Mop descriptions as well, since the semantic descriptions of instructions are similar to TMDLs. However, Cattell addressed several problems found in earlier systems. In particular, he determined that specifying the commutativity of operators should be solved *outside* the description of the target machine, in this case by using axioms. Cattell also observed that: “the machine representation does *not* say how to generate code for the machine in any way” [Cat80].

2.2.5 PO and VPO

Davidson and Fraser use a machine description to achieve machine-independence in their peephole optimizer, PO [DF80, DF84b]. PO’s descriptions use a technique similar to TDML. Consequently, the descriptions take the form of a grammar for syntax-directed translation. One significant difference is that Davidson and Fraser describe the effects of each instruction using ISP-like register transfers, called RTLs (Register Transfer Lists) [DF84b].

Although PO's notation allows storage locations to be named, the only name that has specific meaning is PC, which is used for the program counter. PO assumes that the PC will be incremented after each instruction, therefore, this effect need not be described by each instruction. Davidson makes an observation about the nature of machine descriptions at any level [DF80]:

*“Details irrelevant to the object code may be omitted from the machine description.
... PO does not need to know how the condition code represents comparisons, so the
machine description does not say.”*

Therefore, there are specific details about the machine, such as the condition code representation, that do not affect the interface to the machine. PO's machine descriptions are small; they can be written in an hour or two by someone familiar with the target machine.

Later, Davidson and Fraser developed a compiler—that used PO—for the Y programming language [DF84a]. Combiner¹, a phase of PO, is retargeted using a machine description. However, Combiner does not use the description directly. Instead, the description is translated into a table that Combiner uses. Machine descriptions sometimes require tuning for the compiler to generate good code. Thus, Combiner is not tuned from machine to machine, making it more portable. Combiner does not make time-space trade-offs, so this information is not explicitly encoded in the machine description. Rather, the order of instructions is important in the descriptions. Thus, more specific instructions are placed before their more general, more expensive counterparts. In addition, PO uses a register assignment module that contains tables of information about the register set—a form of register description [DF84a].

Benitez and Davidson have since developed a successor to PO, called *vpo* [BD88, Ben89]. *vpo* uses an improved machine description technique. Many of the problems that Davidson discovered while using PO descriptions have been addressed. Both PO and *vpo* use the machine descriptions to generate recognizers for RTLs [Dav85]. PO descriptions were used to produce finite state automata (FSA) that recognized valid RTLs. Benitez and Davidson refined this method by using Yacc [Joh83] to generate the RTL recognizers [Dav85]. By

1. Combiner is the phase of PO that replaces sequences of register transfer instructions with single instructions that are semantically equivalent [DF84a].

using Yacc-based descriptions, they describe machines with large instruction sets more completely. Furthermore, the PO regular expressions used to generate the FSA's were not powerful enough to describe the assembly language expressions.

In addition to the Yacc-based description, *vpo* uses a formal description of register sets. Register sets are assigned type, size and alignment requirements. Furthermore, the register descriptions allow multiple abstract register sets to be mapped onto the same hardware register set. This provides multiple views of a single register set, which is convenient for machines that use the same registers to store floating-point and fixed-point values.

The *vpo* machine descriptions integrate the techniques used in ISP and PO. From PO, *vpo* takes its Yacc-based description. By using a grammar, common features can be factored and described in a single location. Also, since the description is a Yacc grammar, semantic actions can be used, providing additional flexibility. From ISP, *vpo* takes the register transfer notation (the RTLs) to describe the semantics of instructions. In fact, *vpo* uses RTLs to represent instructions throughout its optimization phases. The notation is a simple, intuitive, application-independent representation of instruction semantics.

Despite the benefits described above, *vpo* descriptions have a number of disadvantages. First, the LALR [ASU86] parsers generated by Yacc are still too restrictive. For some machines, it is difficult to remove reduce-reduce conflicts without compromising the conciseness or readability of the descriptions. Second, the addition of semantic actions, which at first seems beneficial, makes the descriptions more difficult to read since the information is distributed across multiple files. Third, *vpo* provides no formal description of software conventions. Finally, most of the description is still in the form of an implementation—making it less suitable for other applications.

2.2.6 The GNU C Compiler

Using the ideas from PO, The Free Software Foundation's GNU C compiler [Sta92] also uses both RTLs and a machine description to attain retargetability. The machine description is broken into two parts: a set of instruction patterns, and a set of C macro definitions. The macro definitions parameterize the implementation by providing information about the target machine, such as storage layout (*e.g.*, big-endian or little-endian), sizes of supported data types, register usage, and subprogram calling convention. The instruction patterns contain

RTL templates, constraints on the missing pieces of the templates, and an output pattern or C code to generate the assembler output.

The GNU C compiler's machine description is probably the most thorough attempt at parameterizing an implementation. Most aspects of the target machine can be described, in some way, using the instruction patterns and macro definitions. The macro definitions isolate machine-specific details, but do not really *describe* the target architecture in any traditional sense. The technique is not only application-specific, but compiler-specific. Similar to PO, GNU's instruction patterns are used for peephole optimization. Unlike PO, these patterns use a combination of a complex LISP-like syntax for describing the RTL templates, and C code for specifying the format of the assembly output. This combination makes the patterns confusing and difficult to read. It would not be possible to reuse these descriptions since they are so tightly coupled with the implementation of the compiler.

2.2.7 Maril

Bradlee, Henry and Eggers' Marian system [Bra91, BHE91] uses Maril, a machine description language, for describing not only the instruction set, but also the instruction scheduling properties and a limited register description. Maril is the first description system to incorporate details about instruction pipelines. Specifically, associated with each machine instruction are the resources, such as the fetch, decode and execution units, that the instruction requires during each cycle of its execution. This information makes it possible for Marion to use a machine-independent instruction-scheduling algorithm. Additionally, limited information about the register sets can be specified. This includes which registers are volatile or used to pass arguments, which registers are assigned to the frame and stack pointers, which registers hold the return address and return value, and registers that have constant value (*e.g.*, a value of zero).

The Marian system is limited in its use of the description provided by Maril. For example, Marian uses *lcc* [FH91, FH95] to generate code. Since *lcc* has its own code generator, the description is not consulted during code generation; the only portion from the instruction description that is used is the pipeline resource information. Furthermore, it is not clear how effectively Marian uses the information it is given since none of the code generated by the compiler has been run on the target machines.

2.3 Multipurpose Descriptions

As machine descriptions matured, three facts became apparent: 1) machine description systems are difficult to build, 2) machine descriptions are difficult to write and debug, and 3) machine descriptions contain information of interest to all sorts of retargetable applications. As a result, there has been a growing interest in machine description systems that can be used in more than a single application. In this section, we present languages that have the potential of being multipurpose descriptions.

2.3.1 SLED

Ramsey and Fernández's New Jersey Machine Code Toolkit [RF95, RF97] aids in the development of programs that process machine code. The toolkit uses a Specification Language for Encoding and Decoding (SLED) machine code instructions. The toolkit presents the users with an assembly language level of abstraction. Tools that use the toolkit can easily read or emit machine code instructions through a procedural interface.

SLED descriptions concisely specify the binary format of a machine's instructions. From these descriptions, two different procedural interfaces can be generated: an interface that reads machine code and an interface that emits machine code. Using these interfaces, applications can be written that manipulate machine code in a machine-independent manner. The descriptions do not specify how the machine code will be manipulated, but rather the format of machine code.

SLED is a superb example of a description language that can be used for multiple purposes. Although SLED does not describe any other features of machine instructions—including their semantics—SLED provides an effective solution to a difficult problem: describing machine instruction formats. SLED would be a good choice for solving the encoding/decoding problem in a larger system.

2.3.2 λ -RTL

The Zephyr component [ADR98] of DARPA and NSF's National Compiler Infrastructure includes λ -RTL machine descriptions developed by Ramsey and Davidson [RD98a, RD98b]. Since Zephyr uses *vpo* as its optimizer, it must model machine instructions as RTL's. Ramsey and Davidson attempt to formalize *vpo*'s RTL's by using a description language called λ -RTL.

λ -RTL is based on λ -calculus and models a machine's instructions as transformations on the machine's state.

The λ -RTL specification language is still being developed. However, initial descriptions yield insight into the nature of the language. λ -RTL imposes strong types on RTL's. However, because of the underlying formalism, often the λ -RTL processor can infer the types of operations without having to specify them everywhere. This makes the descriptions more compact, while, at the same time difficult to understand without reading the entire description. At this early stage, the effectiveness of λ -RTL has not been evaluated. Unfortunately, the specifications trade readability for conciseness to such a degree that it is not clear that anyone but the specification's author will be able to read them.

2.4 Summary

This chapter presented examples of computer hardware description languages, machine descriptions, and potential multipurpose languages. CHDLs are used in the simulation and synthesis of hardware, while machine descriptions are used in the construction of software. In contrast to their predecessors, multipurpose descriptions attempt to separate what is being described from the description's use.

Although research in the field of computer description systems has been active, no system provides a complete or general solution to the problem. This body of work presents strong evidence that subsequent description systems should address the following problems:

- Retargetable software is difficult to write; so are machine description systems. New description systems should separate a description's form from its purpose.
 - It is difficult to anticipate all the information that all applications may deem necessary. Description systems should be extensible.
 - Descriptions must not only be written, but read. Notation must be familiar to potential authors.
 - Descriptions never seem to be complete. Incomplete descriptions should be usable.
 - Different applications view machines differently. Descriptions must support multiple levels of abstraction and multiple views of a single abstraction.
-

CHAPTER 3

SPECIFYING INSTRUCTION SEMANTICS: CSDL CORE DESCRIPTIONS

In the next three chapters, we develop a framework for building reusable computing system descriptions called CSDL (*Computing System Description Language*). We divide CSDL descriptions into components that are each responsible for describing one feature of a target architecture. In this chapter, we present the CSDL *core* component which is responsible for describing machine characteristics of interest to most, if not all, applications: the target architecture's instruction set.

A core description presents the instruction-set architecture of the machine. This abstraction level consists of the information that is necessary to produce or manipulate instructions for the target machine. We provide this information by defining the effects of instructions on the state of the machine.

Core descriptions are composed of two parts: the semantics of the instruction set and alternative forms or views of instructions such as the assembly language format, the binary encoding of instructions, or the cycle cost of instruction execution. In this chapter, we focus on the formal description of instruction semantics in isolation. Chapter 5 will present how core descriptions may be augmented with whatever additional information an application writer considers necessary.

Our instruction semantics are based on a register transfer notation called register transfer lists (RTLs), so we first present an extant register transfer notation.

3.1 String RTL's

Traditional systems software generates, or operates on, assembly language or binary machine language instructions. Unfortunately, both of these forms of machine instructions vary from machine to machine. For example, to perform a 32-bit signed addition on the MIPS [KH92], the assembly language instruction is:

```
add r1, r1, r2
```

while on the Motorola 68020 [Mot85] the assembly form is:

```
add d2, d1
```

these two instructions differ not in their semantics, but rather in their concrete syntax. One reason this difference occurs is that each assembler defines the format of lexical tokens (*e.g.*, opcode mnemonics, registers, constants, and addressing modes) and the ways in which they may be combined (the assembly language). The result is that it is impossible to determine without knowing the particular assembly language whether the instruction:

```
add r2, r3, r1
```

adds registers two and three and places the result in register one, or if it adds registers three and one and stores the result in register two.

Such trivial machine dependencies, as well as far less trivial differences, can be eliminated by expressing the semantics of instructions using register transfers, or RTL's (Register Transfer Lists). One dialect of register transfers that is representative of the technique was developed by Davidson and Benitez [BD88]. This form, which we call "string RTL's," is presented in this section. RTL's make it possible for software to eliminate trivial syntactic differences and concentrate on semantic differences that reflect each machine's capabilities at the machine instruction level.

A highly successful method of eliminating machine dependencies is to express each machine instruction in a language whose semantics are invariant across platforms. Instructions are then manipulated in this language by systems software whose algorithms are machine independent.

3.1.1 String RTL Syntax and Semantics

String RTL's are composed of registers, memory references, constants, labels, local and global identifiers, macros, and operators. We briefly describe the syntax of each of these tokens here.

3.1.1.1 Registers

Registers are represented using the notation:

$$r[num]$$

where r is a lower-case letter that indicates the type of value the register currently holds. num is a decimal number that indicates which register is being referenced. For example, $b[5]$ typically designates the sixth register. This register holds a byte (thus, the b register type).

3.1.1.2 Constants

Constants are always positive and can be either integer or floating-point. Integer constants are strings of decimal digits. Floating-point constants use the notation:

$$mantissa E sign exponent$$

where $mantissa$ is a string of decimal digits representing the integer value of the mantissa, $exponent$ is a string of decimal digits representing the value of the exponent, and $sign$ is either '+' or '-' to indicate the sign of the exponent. Negative constants can be obtained by applying the unary negation operator ('-') to the constant.

3.1.1.3 Operations

Register transfers not only transfer data from one location to another, they also perform various arithmetic and logical operations. The set of operations is limited to 36 built-in unary and binary operators that are available on most architectures. This includes the standard arithmetic operations such as addition and subtraction of signed integers (denoted '+' and '-', respectively), bitwise logical operations such as AND, OR, and NOT ('&', '|', and '~'), and relational operations such as less than, greater than, and equal ('<', '>', and ':'). Finally, an RTL effect is not complete without the assignment operator ('=') which performs a store operation (thus the use of ':' for relational equal). The RTL effect:

$$r[1]=r[2]+r[3]; \tag{3-1}$$

denotes that register two is added (using signed integer arithmetic) to register three with the result being placed in register one. The semicolon (;) marks the end of the effect. RTL operators are often overloaded, and the type of operation is determined by the type of the operands. For example, addition of two registers that contain floating-point values could be described using the effect:

$$f[1]=f[2]+f[3];$$

Thus, the same operator, '+', is used to designate two different operations: signed-integer addition and floating-point addition.

3.1.1.4 Macros

Often times, it is necessary to extend the set of built-in operations. This is accomplished using RTL function macros. A function macro is represented by an identifier composed of exactly two uppercase letters followed by a comma-separated list of expressions enclosed in brackets. A common use of function macros is to perform type conversions. For example, to convert a floating-point value into an integer value, one could introduce the `FI` macro:

```
r[5]=FI[f[5]];
```

The meaning of function macros is machine-dependent and is thus undefined by the notation. Their meaning must be implicitly understood by the algorithms that manipulate them.

Function macros are also used to abstract away the details of complex instructions. For example, the `SAVE` instruction on the SPARC [Sun87] that provides a new register window is described using the `SV` function macro:

```
r[14]=SV[r[14]+64];
```

This effect only indicates that `r[14]` is both read and written. The details of which registers are saved, and which registers change values because the register window has moved, remain unspecified.

In addition to function macros, string RTL's also allow for macros to be used to describe special storage locations in the target machine. Examples of these include `PC` and `CC` which designate the program counter and condition codes respectively.

3.1.1.5 Memory

Memory references are represented using the notation:

```
M[address]
```

where `M` is an uppercase letter that indicates the type of value the memory location currently holds. `address` is an arbitrary RTL expression that indicates the address of the memory location being referenced. The RTL expression for a memory fetch using register displacement is:

```
F[r[4]+12]
```

In this case, the result of this expression is likely a single-precision floating-point value¹. Unlike register indices, memory addresses may use arbitrarily complex expressions to represent the necessary addressing mode. Table 3-1 shows several of the most commonly known addressing modes.

RTL Expression	Addressing Mode
R[_global_id]	memory direct
R[w[4]]	register indirect
R[w[4]+12]	displacement
R[w[4]*4]	scaled
R[(w[4]*4)+12]	scaled displacement
R[w[4]+w[7]]	indexed
R[R[_global_id]]	memory indirect

Table 3-1. Sample RTL address expressions (excerpted from [Ben94]).

3.1.1.6 Symbolic Addresses

String RTLs use three types of expressions to name memory addresses and constants symbolically. They are: labels, global identifiers, and local identifiers. Labels most commonly mark the target of a branch instruction. Labels are designated using the character 'L' followed by a decimal number (*e.g.*, L15). Global identifiers mark constant address values and function entry points. A global is represented using a string of letters and digits (*e.g.*, index0). Local identifiers usually represent constant offset values (typically from the stack pointer). A local is represented using a string of letters and digits followed by a period (*e.g.*, i.).

Locals and globals, as well as other tokens, have an encoded string variation as their internal representation. The internal form uses two bytes to compactly store which symbol is referenced in the RTL. These two bytes are used as the key for a symbol table to quickly access all necessary symbol information, such as a symbol's offset.

3.1.1.7 Instruction Effects

Ultimately, the purpose of string RTLs is to describe the effect a machine instruction has on the state of the target machine. This is achieved by combining the various string RTL expres-

1. Although there are conventions regarding the meaning of memory types, their meanings are machine dependent.

sions described above (and summarized in Table 3-2) into a list of instruction effects. An effect contains a single assignment operation on some storage location. In many cases, as shown above, instructions can be described using a single effect. Instructions that modify more than a single location are described using multiple string RTL effects. For example, on many machines, addition also sets a condition code register. This would be expressed using:

$$r[4]=r[4]+r[3];CC=(r[4]+r[3])?0;$$

where the assignment to the macro `CC` describes the instruction's effect on the machine's condition codes. All expressions are assumed to be evaluated before any assignments are made.

Type	Regular Expression ^a	Example
Register	[a-z][[0-9]⁺]	<code>r[5]</code>
Integer Constant	[0-9]⁺	<code>15</code>
Floating-point Constant	[0-9]⁺E[+ -][0-9]⁺	<code>15E10</code>
Operation	<i>expr^b op^c expr</i> or <i>op expr</i>	<code>r[1]+5</code>
Macro	[A-Z][A-Z]	<code>PC</code>
Function Macro	[A-Z][A-Z][<i>expr, ...</i>]	<code>FI[f[5]]</code>
Memory Reference	[A-Z][<i>expr</i>]	<code>R[r[14]+12]</code>
Local identifiers	[A-Za-z0-9_]⁺.	<code>i.</code>
Labels	L[0-9]⁺	<code>L15</code>
Global identifiers	[A-Za-z0-9_]⁺	<code>_main</code>
<i>effect</i>	<i>expr = expr;</i>	<code>r[5]=12;</code>
RTL	<i>effect⁺</i>	<code>r[1]=r[2];r[1]=r[2]</code>

Table 3-2. Summary of formats for string RTL expressions.

- Tokens are described using extended regular expressions. Literals are displayed in bold.
- expr* is any RTL expression defined in the table.
- op* is any one character RTL operator such as +, -, *, /, <, >, etc.

3.1.1.8 Transfers of Control

Instructions that perform transfers of control use several different formulations. Both conditional and unconditional branches are described by assigning to the program counter. The string RTL:

```
PC=L43;
```

describes an unconditional branch to the instruction labeled L43. Conditional branches use the relational operators to compute the target address:

```
PC=CC:0,L43;
```

In this case, the PC is set to L43 only if the value of the condition codes equals zero. Otherwise, PC is not set by this effect. The list operator (',') is used to augment the assignment operator to designate conditional assignment.

The two other common forms of transfer of control are procedure call and return. Procedure calls are represented by assignment to the ST macro:

```
ST=_doit;
```

and returns are described as:

```
PC=RT;
```

which indicates that control is transferred back to the address found at the top of the call stack. Although the effect of a procedure is to set the program counter, the string RTL effect for procedure call sets the special macro ST instead of PC. This makes it possible to quickly distinguish procedure calls from branches in string RTL's. Finding procedure calls quickly is important since procedure call sites are of interest for many analyses including building program call graphs. The procedure return effect has a special form for similar reasons.

3.1.2 Analysis and Manipulation

With a firm understanding of the syntax and semantics of string RTL's, we can now discuss how software that uses string RTL's can analyze and manipulate machine-dependent information in a machine-independent way.

First, assume that there are ways to convert an assembly language program into a semantically equivalent sequence of string RTL's and *vice versa*. Both translations can be easily achieved using syntax directed translation¹ [ASU86]. Given these translations, it is common

1. Benitez's VPO optimizer [BD88] in fact uses syntax directed translation to convert the string RTL's it generates into assembly language before the result is assembled.

to think of a program that is expressed as a sequence of RTL's as an assembly language program without the obvious shortcomings that such a machine-dependent notation has.

String RTL notation has been applied to a wide variety of systems software applications that traditionally manipulate or generate assembly or machine language instructions. These include compilers, optimizers, linkers, and programs that perform program instrumentation [DF80, BD88, Wha90, Sta92]. In this section, we briefly detail how RTL's are used to achieve simple program transformations in an optimizer. However, the ideas are equally applicable to any other application that works with machine language instructions.

Probably the single most important aspect of string RTL's is that they make the sets and uses of registers and memory locations explicit. This makes it easy to identify data dependencies in sequences of instructions. For example, given the following sequence of instructions:

```
r[1]=r[2];
r[1]=r[2]+r[3];
```

it is trivial to identify that the first instruction is useless since the second instruction immediately writes (sets) over the result of the first instruction. Therefore, the first instruction may be harmlessly deleted without changing the semantics of the sequence.

More commonly, RTL's are used to identify where multiple RTL's can be combined, similar to peephole optimization, into a single RTL or a shorter sequence of RTL's. For example, in the RTL sequence:

```
r[1]=r[2];
r[3]=R[r[14]+12];
r[4]=r[1]+r[3];
```

(3-2)

we can substitute the expression $r[2]$ for $r[1]$ and $R[r[14]+12]$ for $r[3]$ in the third RTL to yield the RTL:

```
r[4]=r[2]+R[r[14]+12];
```

(3-3)

This RTL describes a new effect. If this new effect is performed by an instruction on the target machine, then the three-instruction sequence of (3-2) may safely be replaced with (3-3). The true benefit of this transformation is not realized until the first two RTL's are removed from the sequence. This may occur if there are no more uses of the current values of $r[1]$ and $r[3]$.

Benefits may also be reaped through algebraic manipulation of RTL expressions. The RTL:

$$r[1]=r[2]*8;$$

uses multiplication which is often an expensive operation. In this special case, we can replace this with the cheaper RTL:

$$r[1]=r[2]\{3;$$

where '{' denotes signed shift-left.

Finally, because sets and uses are explicit in RTL's, it is possible to write general, machine-independent algorithms to reorder sequences of RTL's into more efficient sequences that are semantically equivalent. This is often performed in the presence of memory references. Here is a sequence of RTL's that contains two memory references to local variables i and j :

$$r[7]=r[7]+1;$$

$$r[6]=R[r[14]+i.];$$

$$R[r[14]+j.]=r[7];$$

$$r[6]=r[6]\{2;$$

Because the second RTL does not use $r[7]$ and the third RTL does not use $r[6]$, these two RTL's can be exchanged yielding a sequence of RTL's in which the lifetime of $r[6]$ does not overlap the lifetime of $r[7]$. Therefore, all instances of $r[6]$ can be replaced by $r[7]$, yielding the following sequence of RTL's:

$$r[7]=r[7]+1;$$

$$R[r[14]+j.]=r[7];$$

$$r[7]=R[r[14]+i.];$$

$$r[7]=r[7]\{2;$$

Although this sequence of RTL's is no shorter, and uses the same operations, the sequence uses one fewer registers. The unused $r[6]$ can then be used in other locations to reduce the number of memory references. This, in turn, will improve the overall quality of the code.

Each of these examples are transformations that are performed by typical optimizers including those that do not use RTL's. The important difference is that the algorithms that perform these RTL transformations need only be written once rather than again and again for each new assembly language and target machine. When a new target machine is introduced, the translations to and from its assembly language must be made. This, however, is signifi-

cantly easier than rewriting and debugging all the algorithms that manipulate the target machine's instructions. Finally, perhaps the greatest benefit of string RTL's is their natural human-readable form. All of the above manipulations were described independently of any particular architecture. RTL's are an excellent medium for discussing machine-dependent instruction manipulations without the burden of presenting a new assembly language notation for each machine.

3.2 τ RTL's

Using RTL's to specify the semantics of instructions has significantly improved the retargetability of systems software that use them. However, experience with string RTL's has revealed a number of shortcomings that prevent software from exploiting the full potential of the RTL concept. These shortcomings are, in some cases, so severe that a complete reworking of the notation was warranted. In this section, we present τ RTL's, a new RTL form that addresses these concerns.

τ RTL's differ from string RTL's in three fundamental ways: their type system, syntax, and underlying representation. At first glance, the most noticeable change is the use of an extended character set and formatting for the concrete syntax of τ RTL's. We will discuss these features in Chapter 5. Presently, we describe the fundamental differences that enable more effective use of RTL's in building machine-independent software.

3.2.1 Syntax

Before discussing the conceptual differences between string RTL's and τ RTL's, we first present the concrete syntax of each τ RTL expression.

3.2.1.1 Constants

The simplest expressions are integer and floating-point constants. Integer constants have the form:

[*-*] *digits*

where *digits* is a string of decimal digits, optionally preceded by a minus sign. Floating-point constants have the form:

[*-*] *mantissa sign exponent*

where *mantissa* is a string of decimal digits representing the integer value of the mantissa, *exponent* is a string of decimal digits representing the value of the exponent, and *sign* is either '+' or '-' to indicate the sign of the exponent.

3.2.1.2 Types

New with τRTL's is the type expression. A type is specified using the notation:

$$i, \textit{size}$$

where *i* is a subscripted single letter specifying the interpretation (such as signed or floating-point) of the value being typed, and *size* is a subscripted string of decimal digits indicating the size of the value in bits. Types are used to indicate the interpretation and size of intermediate values.

3.2.1.3 Operations (Typed Expressions)

Operations take zero or more operands and produce a single result. The types of each operand and result must be specified using a type expression. Operations using binary infix operators are written as:

$$(\textit{expr op expr}) \textit{type}$$

where *expr* is a type-decorated expression, *op* is a single character built-in operator, and *type* is a type specifying the type of the result. Operations may also use the prefix form:

$$\textit{func} (\textit{expr}, \dots) \textit{type}$$

where *func* is either a built-in operator or a string of letters. Using either form yields another type-decorated expression.

3.2.1.4 Storage

Storage expressions represent fetches and stores to a machine's memory—either primary memory or registers. There are three forms of storage expressions:

$$\textit{name} \tag{3-4}$$

$$\textit{name} [\textit{expr}] \tag{3-5}$$

$$\textit{name} [\textit{expr}, \textit{size}] \tag{3-6}$$

where *name* is a string of one or more letters (typically just one) that names the storage location, *expr* is a type-decorated expression denoting the index, and *size* is a string of digits representing the number of cells referenced. For individual locations, such as the program counter, or condition codes, form (3-4) is used. For primary memory and register sets that

appear as arrays of cells, form (3-5) is used for individual cell references, while form (3-6) is used for contiguous multi-cell references (*e.g.*, as multi-byte memory fetches).

In addition to basic storage cell references, cells may be concatenated together using:

(*storage* : *storage*)

where *storage* is any basic storage expression. Sub-cell references may also be made by using the bit extraction expression:

storage^{*high..low*}

where *storage* is any basic storage expression and *high* and *low* are strings of superscripted digits specifying the highest and lowest bit locations to be extracted.

3.2.1.5 Instruction Effects

As with string RTLs, an instruction's effect on storage is specified using a list of effect expressions. The syntax for an effect is

storage \leftarrow *texpr* ;

where *storage* is a storage expression and *texpr* is a type-decorated expression. When more than one effect is included in an list, each right-hand expression is evaluated before any assignment is made to left-hand storage locations.

3.2.1.6 Syntax Summary

In summary, τ RTLs are composed of the following kinds of expressions:

- constants (integer, or floating-point),
- memory fetches (registers or primary memory locations),
- operators, and
- memory stores.

We summarize the syntax for τ RTL expressions in Figure 3-1 using a context free grammar.

Unlike string RTLs, all τ RTL expressions and sub-expressions have explicit types.

Conversely, storage locations are not typed. We think of this difference as moving the types away from the storage locations towards the operators. For example, the simple register-register addition from (3-1) would be expressed in τ RTL as:

$r[1_{u,5}]_{s,32} \leftarrow r[2_{u,5}]_{s,32} + r[2_{u,5}]_{s,32}$;

If you remove all the type expressions, which are indicated using subscripts, the result looks very much like its string RTL equivalent. These type expressions are the subject of the next section.

1.	<i>RTL</i>	→	<i>effects</i>
2.	<i>value</i>	→	int float <i>storage_expr</i> <i>operation</i>
3.	<i>type</i>	→	letter , int
4.	<i>typed_value</i>	→	<i>value type</i> <i>typed_operation</i>
5.	<i>storage_expr</i>	→	<i>storage_expr</i> ^{int.int}
6.			(<i>storage_expr</i> : <i>storage_expr</i>)
7.			<i>storage</i>
8.	<i>storage</i>	→	letters [<i>typed_value</i>]
9.			letters [<i>typed_value</i> , int]
10.			letters
11.	<i>operation</i>	→	<i>typed_value op typed_value</i>
12.			<i>func</i> (<i>arg_list</i>)
13.	<i>typed_operation</i>	→	(<i>typed_value op typed_value</i>) <i>type</i>
14.			<i>func</i> (<i>arg_list</i>) <i>type</i>
15.	<i>arg_list</i>	→	
16.			<i>typed_value typed_values</i>
17.	<i>typed_values</i>	→	, <i>typed_value</i>
18.		→	, <i>typed_value typed_values</i>
19.	<i>func</i>	→	op letters
20.	<i>effect</i>	→	<i>storage_expr</i> ← <i>typed_value</i>
21.			<i>storage_expr type</i> ← <i>value</i>
22.	<i>effects</i>	→	<i>effect</i> ;
23.			<i>effect</i> ; <i>effects</i>

Figure 3-1. Context-free grammar for τ RTL's^a.

- a. **int**, **letter**, **letters**, and **op** are grammar terminals described by the regular expressions [0-9], [A-z], [A-z]+, and [+<?...], respectively.

3.2.2 τ RTL Types

Each τ RTL expression is labeled with a type. τ RTL types have a type signifier and size. These are denoted using subscripts. For example, the expression

$$15_{u,13}$$

indicates that the integer constant 15 is represented using a 13-bit unsigned integer representation. Both type and size are necessary since the constant expression 15 indicates neither the number of bits used to represent the number, nor the representation (unsigned or two's complement). Currently, our descriptions use four type-signifiers, although additional ones can be added at any time. They are:

- u (unsigned integer)
- s (signed two's complement integer)
- f (floating-point)
- b (bitstring)

Using these types, we can build up larger τ RTL expressions. A simple register ADD instruction from the MIPS can be expressed as follows:

$$r[1_{u,5}]_{s,32} \leftarrow r[2_{u,5}]_{s,32} + r[3_{u,5}]_{s,32}; \quad (3-7)$$

This yields significantly more useful information than string RTL's. First, from the register numbers' type, we know that register indices may not exceed 31. Second, we know that each of the three registers is capable of holding a 32-bit value. An important feature of the notation that we use for τ RTL's is locations are not typed, but rather intermediate values and operators are typed. Consequently, the type on the left-hand side (left of ' \leftarrow ') of (3-7) does not apply to the register $r[1_{u,5}]$ but rather to the result of the addition operation on the right-hand side.

The same is true of the registers on the right-hand side. Thus, each of the three $s,32$ types designate the type of addition being performed rather than the type of the registers. Therefore, we know that the machine can perform the addition of two signed 32-bit numbers, with an signed 32-bit number as its result (we denote this $+_{s,32} \times s,32 \rightarrow s,32$).

To some, it may seem strange to place the types near the operands (or more strangely near the result location) rather than near the operators to which they belong. Here are two other formulations we considered:

$$r[1_{u,5}] \leftarrow r[2_{u,5}] +_{s,32} \times s,32 \rightarrow s,32 r[3_{u,5}]; \quad (3-8)$$

$$r[1_{u,5}] \leftarrow (r[2_{u,5}]_{s,32} + r[3_{u,5}]_{s,32})_{s,32}; \quad (3-9)$$

Formulation (3-8) is quite cumbersome. The operator's type significantly increases the distance from operator to operand. As the number of operators in the expression increases, the expressions become more and more unwieldy. Formulation (3-9) moves the type of the operator's result closer to where the result is produced. It also has the undesirable effect of requiring the addition of parenthesis to separate the result type from the type of the rightmost operand. However, this form is used when we include more than a single infix operand, such as:

$$r[1_{u,5}]_{s,32} \leftarrow (r[2_{u,5}]_{s,32} + 15_{s,32})_{s,32} - r[3_{u,5}]_{s,32};$$

however, here the parenthesis are also needed to determine the order of expression evaluation.

The inclusion of types for register indices may also seem like a strange formulation. However, this is just a specialization of a more general method of specifying addressing modes using τ RTL. A great advantage of string RTL's is their very general treatment of computational expressions. Whether a computation represents the result of an instruction execution, or is just a calculation of an address computation, the expression is the same. However, string RTL's treat register indices specially. Register indices may only be integer constants. In τ RTL, we treat registers as any other type of storage. Therefore, arbitrary expressions may be used as

register index expressions. Since the indices of memory references have types (as much a result of the operators in the memory address calculation as anything) the analogous form for a reference to register 14 is the same:

$$r[14_{u,5}]$$

3.2.3 Aliasing

A common problem with string RTLs, as well as other instruction description techniques, is memory aliasing. In string RTLs, aliasing occurs in two ways:

1. Memory aliasing due to typing of storage, and
2. Memory aliasing due to multiple-unit fetches.

In string RTLs, the type of a memory location is encoded in the location's name. So, when register one contains a byte, its name is $b[1]$. When register one contains a long, its name is $r[1]$. This convention makes it easy to keep track of what kind of value is held in the register. However, it makes it difficult to identify that $r[1]$ and $b[1]$ are, in fact, the same register. An optimizer that uses string RTLs will consider these two registers to be distinct. This wreaks havoc when the following sequence of string RTLs are incorrectly generated¹ by a code generator:

```
b[1]=255;
r[2]=r[2]&r[1];
```

In this case, register $r[1]$ switches types between the two RTLs. This will cause an optimizer to remove the first RTL because it appears to be useless code ($b[1]$ is set but never used). This will cause a error in the resulting code because $b[1]$ may not contain the correct value. Although there is an error in the compiler's implementation (the wrong RTL was generated), the memory aliasing makes it difficult or impossible for the error to be automatically detected.

A second common source of memory aliasing involves multi-addressable-unit references. Take, for example, the following sequence of string RTLs:

```
r[2]=R[r[14]+18];
R[r[14]+16]=r[1];
```

1. Although this example occurs when there is a bug in the code generator, such bugs commonly happen and must be tracked down. Furthermore, such bugs are, in fact, *caused* by the awkward typing that string RTLs impose on storage locations. The programmer knows that $b[1]$ and $r[1]$ are the same, but the notation makes it appear to the software that they are different.

When an optimizer examines this sequence, it may wish to move the first RTL down past the second to reduce the lifetime for register $r[1]$ (or register $r[2]$ for that matter). This appears to be safe since the load in the first RTL appears to be from a different location than the store in the second RTL. However, because both the load and store are 32-bit data instructions, they reference multi-byte quantities causing the two memory references to overlap (they both reference offsets 18 and 19 off of $r[14]$), as shown in Figure 3-2. If the first RTL is moved past the second, these values may change due to the store in the second RTL. Such transformations are common in many optimizations and require specialized machine-dependent logic in the algorithms to recognize and handle this case correctly.

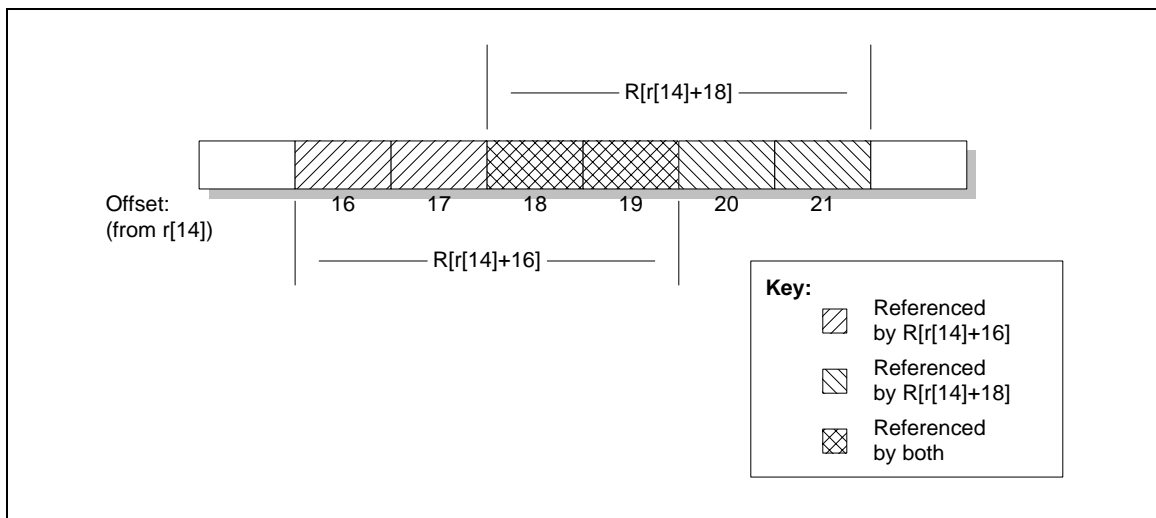


Figure 3-2. Memory aliases created by overlapping memory references.

From the two examples above, it is clear that we would like to eliminate the opportunities for aliasing of locations. By removing aliasing as a problem, we can significantly reduce the problems that applications have when performing various analyses on τ RTL's. We eliminate the first form of aliasing by not typing (or giving multiple names to) registers and memory locations. The second form of aliasing is eliminated by requiring that all memory addresses be explicit. The concrete syntax for the above memory location is:

$$m[(r[14_{u,5}]_{u,32} + 16_{u,32})_{u,32}, 4]$$

The '4' at the end of the index refers to the number of addressable units that the address is referencing and is required for multi-cell memory references. From the address and the size, the addresses are automatically expanded to include locations 16, 17, 18, and 19 in the abstract form of the τ RTL's.

3.2.4 Notation

Although the addition of extended characters and a new type system make τ RTL's appear quite different from string RTL's, underneath, they share many common features:

- instructions are still described using lists of effects,
- address computations are expressed using arbitrary τ RTL subexpressions,
- storage location reads and writes are explicit, and
- instructions that transfer control assign to the program counter storage location.

There are a few differences outside the type system and memory aliasing that make the τ RTL notation easier to read for humans and easier to manipulate for machines. There is no better example than the subtle interaction of types and operator overloading.

String RTL's overload operators to yield natural expressions like:

$$r[1]=r[2]+r[3];$$

and

$$f[1]=f[2]+f[3];$$

These two string RTL's use the same operator, but perform two different operations (integer versus floating-point addition). Unfortunately such overloading is only as flexible as the type system. String RTL types describe size (*i.e.*, byte, word, and long), and in some cases encoding (*i.e.*, long versus float which are different encodings but the same size). Operations that use identical storage types but different representations, such as signed- and unsigned-integers, cannot simply use operator overloading to distinguish operations. For example, there are signed and unsigned versions of many of the relational operations. The string RTL expression

$$r[1]=r[2]>r[3]; \tag{3-10}$$

compares the registers $r[1]$ and $r[2]$ using signed greater-than. To perform unsigned greater-than, a different operator must be used since the same RTL as (3-10) would be formed if the ' $>$ ' operator were used. We must select a new operator to distinguish the two different operations on the same storage type. Unsigned less-than is formed using the ugly ' h ' operator (' g ' is used for greater-than-or-equal):

$$r[1]=r[2]hr[3];$$

Perhaps the most ridiculous formulation is:

$$r[1]=r[2] r[3];$$

where the space operator (' ') indicates unsigned shift left! In τ RTL, each of these operations is cleanly described using the same operator and different types on values being extracted from the storage locations:

$$r[1_{u,5}]_{s,32} \leftarrow r[2_{u,5}]_{s,32} > r[3_{u,5}]_{s,32};$$

$$r[1_{u,5}]_{u,32} \leftarrow r[2_{u,5}]_{u,32} > r[3_{u,5}]_{u,32};$$

Spaces may also be used to separate the operators from the operands since the space character is not an operator in τ RTL.

The improved type system and extended character set change the formulation of some string RTL operations. Other string RTL operators are not replaced, but instead deleted. For example, the compound operator AND NOT ('b') is deleted and formed in τ RTL by the composition of the built-in AND and NOT operators. So, the string RTL:

$$r[1] = r[2] \text{br} [3];$$

becomes

$$r[1_{u,5}]_{b,32} \leftarrow \neg(r[2_{u,5}]_{b,32} \wedge r[3_{u,5}]_{b,32})_{b,32};$$

The changes in built-in operators from string RTLs to τ RTLs are summarized in Table 3-3.

3.2.5 Abstract Syntax

From the preceding sections it may appear that τ RTLs are just another string representation of RTLs. This is intentional. We designed the concrete syntax of τ RTLs to be intuitive and natural for the programmer. Since this was one of the strengths of string RTLs we included it in our design. However, the internal representation, or abstract syntax, is not based on strings as string RTLs are. Instead, τ RTLs are represented internally using trees (τ stands for tree). This distinction is important because, although RTLs are often viewed by the programmer, their primary role is to present machine-dependent information in a machine-independent form for manipulation by programs.

Using trees as an internal representation has many benefits. First, all τ RTL subexpressions are τ RTL subtrees which permits the replacement of one subexpression by another. This facilitates a common RTL operation: forward substitution. Second, once in tree form, the order of evaluation of subexpressions is explicit. Third, since trees use pointers, it is possible for two or more τ RTLs to share common subtrees, or common subexpressions. Fourth, unlike strings, trees provide faster than linear-time access to subtrees (in strings, to find the right-

String RTL Character	τ RTL Operator	Description
' ' (space)	\leftarrow	Left shift, unsigned
!	\neq	Not equal, signed and unsigned
"	\Rightarrow	Right shift, unsigned
#	mod	Modulus, unsigned
\$	Δ	Sign extend
%	mod	Modulus, signed
&	\wedge	Bitwise AND
'	\leq	Less than or equal to, signed
*	\times	Multiplication, unsigned
+	+	Addition
,	unused	List separator
-	-	Subtraction, unary minus
/	\div	Division, signed
:	\equiv	Equal, signed and unsigned
;	;	RTL separator
<	<	Less than, signed
=	\leftarrow	Assignment
>	>	Greater than, signed
?	unused	Compare, signed
@	\times	Multiplication, unsigned
\	\div	Division, unsigned
^	\oplus	Bitwise XOR
`	\geq	Greater than or equal to, signed
b	Synthesized	Bitwise AND NOT
d	Second effect	Auto-decrement
g	\geq	Greater than or equal to, unsigned
h	>	Greater than, unsigned
i	Second effect	Auto-increment
l	<	Less than, unsigned
o	Synthesized	Bitwise OR NOT
s	\leq	Less than or equal to, unsigned

Table 3-3. Built-in RTL operator summary .

String RTL Character	τ RTL Operator	Description
u	unused	Compare, unsigned
x	Synthesized	Bitwise XOR NOT
{	\Rightarrow	Left shift, signed
	\vee	Bitwise OR
}	\Leftarrow	Right shift, signed
~	\neg	Unary negate

Table 3-3. Built-in RTL operator summary (*Continued*).

hand side of an RTL, the left-hand side must be scanned). This should improve the performance of many algorithms. Fifth, as we will see shortly, string RTLs rely on string parsing techniques, specifically, LALR (Yacc) grammars. Trees free us from this restriction and promote the use of quick tree matching techniques [AGT89, FHP92]. Sixth, in contrast to strings, it is not possible to build a malformed tree. Properly formed trees that are illegal (due to the type system) can easily be detected using machine-independent algorithms. Finally, during the process of changing τ RTL token strings into trees, many errors can be detected because only a subset of τ RTL token strings correspond to properly formed τ RTL trees.

In addition to all of the above benefits, τ RTLs make fetching and storing of storage locations explicit. The trees are also strongly typed. For example, the tree shown in Figure 3-3 attempts to describe a memory load using displacement addressing. In this case, the `s,16` type attached to the displacement constant 15 does not match the corresponding `u,32` operand type in the addition operation. Such errors can be detected because each subtree must have a type and each operator must specify the types of its operands. In this case, the type conflict is due to a missing conversion operation. A properly typed tree for this instruction is shown in Figure 3-4.

This load example illustrates that since τ RTLs are strongly typed, explicit type conversions are required. Although it is possible to infer that the type conversion is happening, we choose to insist that the conversion be explicit. This makes it possible to easily identify common errors in instruction semantics. It is not uncommon to accidentally forget to put a conversion in place, thus creating an RTL with the improper semantics.

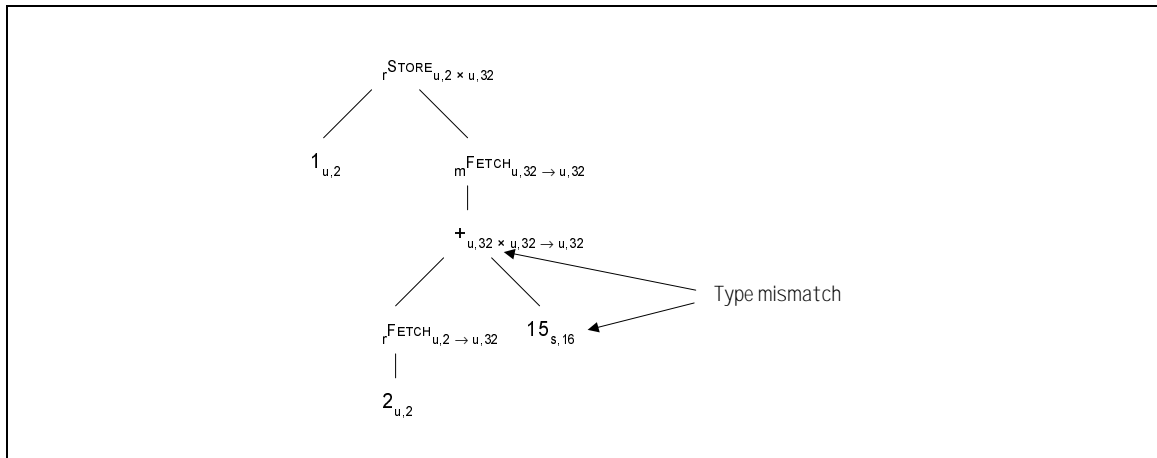


Figure 3-3. Improperly typed τ RTL for a load.

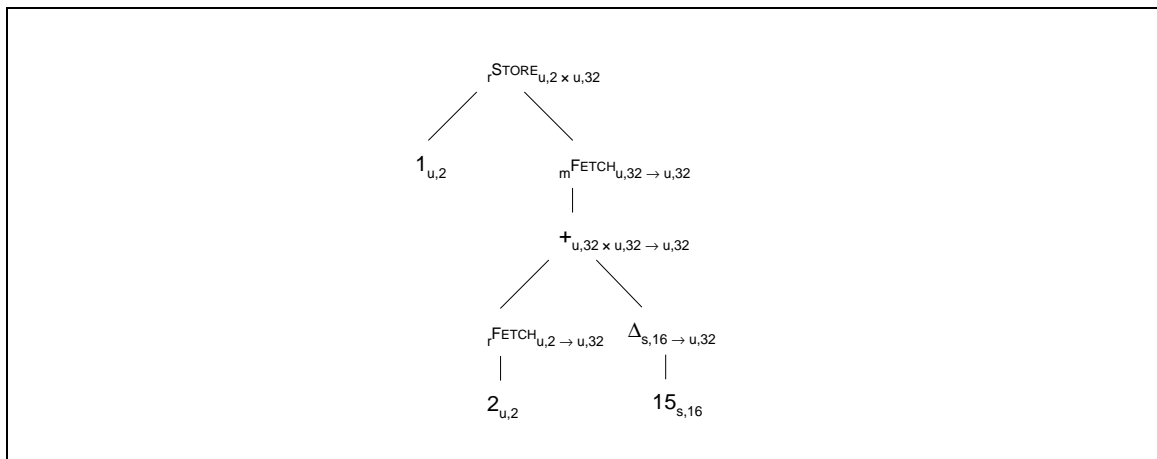


Figure 3-4. A properly typed τ RTL for a load.

Using the τ RTL types also makes it possible to automatically identify common errors in manipulation of trees. For example, consider the two τ RTL sequence:

$$r[1_{u,5}]_{u,32} \leftarrow r[2_{u,5}]_{u,32} + \Delta(15_{u,15})_{u,32}; \quad (3-11)$$

$$r[3_{u,5}]_{b,32} \leftarrow m[(r[1_{u,5}]_{u,32} + \Delta(5_{u,9})_{u,32})_{u,32}, 4]; \quad (3-12)$$

The τ RTL trees corresponding to these instructions are shown in Figure 3-5. Using forward substitution, we can replace the fetch of $r[1_{u,5}]$ in (3-12) with the right-hand side of (3-11).

The resulting subtree is shown in Figure 3-6 and corresponds to the subexpression:

$$((r[2_{u,5}]_{u,32} + \Delta(15_{u,15})_{u,32})_{u,32} + \Delta(5_{u,9})_{u,32})_{u,32} \quad (3-13)$$

This subtree can then be algebraically simplified to combine the two constants. In doing so, a type must be chosen for the resulting constant (20). There are two obvious choices: $u,9$ and $u,15$. Choosing the type $u,15$ could result in the subtree pictured in Figure 3-7. Since the type for the constant ($u,15$) and the type for the convert (Δ) operand ($u,9$) differ, we have an incor-

rectly constructed τ RTL subtree. Such errors can easily be detected by a general tree type-checker.

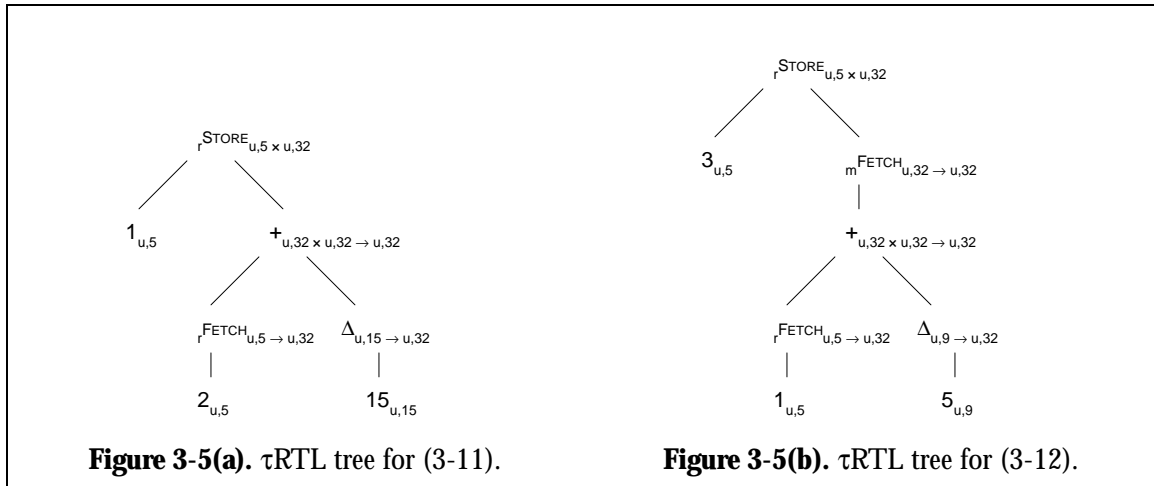
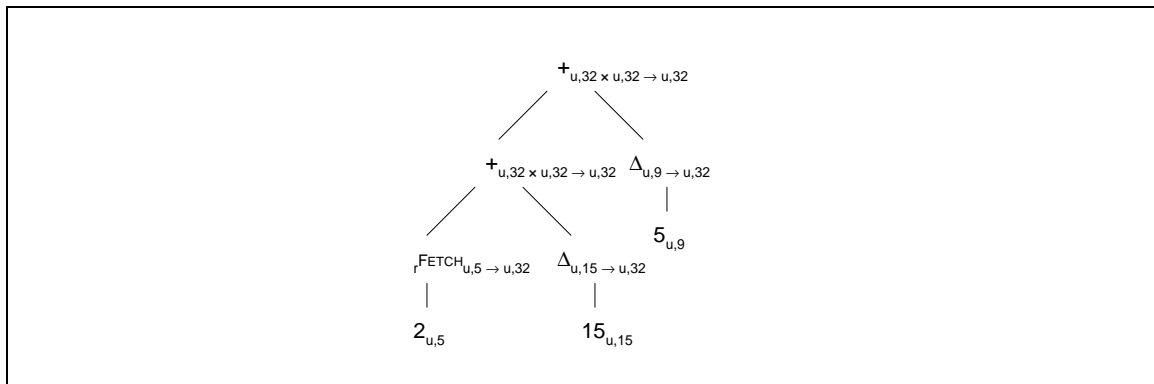
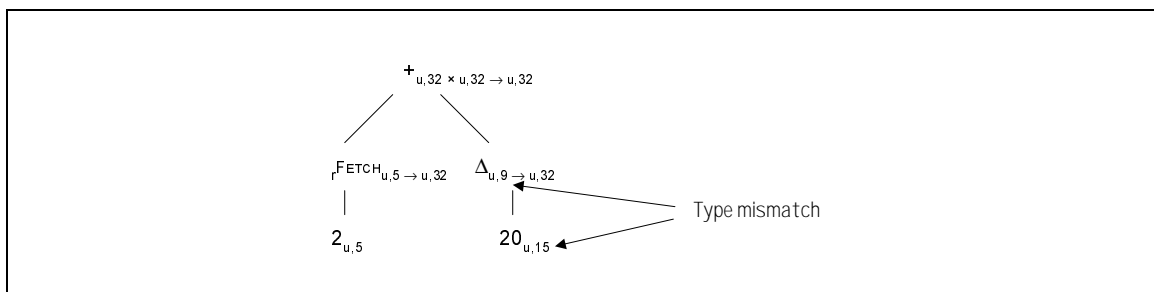
Figure 3-5(a). τ RTL tree for (3-11).Figure 3-5(b). τ RTL tree for (3-12).Figure 3-5. Abstract syntax for two τ RTLs.

Figure 3-6. Combined subexpression.

Figure 3-7. Incorrect simplification of $(r[2_{u,5}]_{u,32} + \Delta(20_{u,9})_{u,32})_{u,32}$ tree.

Clearly, using trees as the abstract syntax provides significant improvement over the string-based approach used previously. Although trees are an excellent representation for machine manipulation, they are not a natural form for humans to use. Fortunately, τ RTLs use

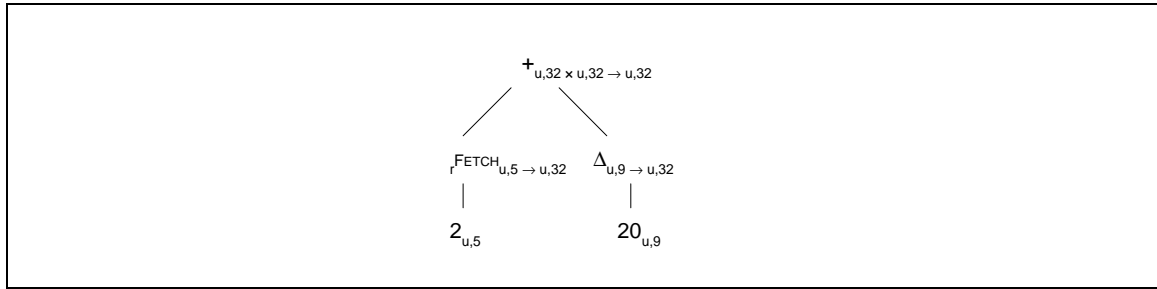


Figure 3-8. Correct simplification of $(r[2_{u,5}]_{u,32} + \Delta(20_{u,9})_{u,32})_{u,32}$ tree.

a natural string concrete syntax freeing the programmer from having to think of τ RTL's as trees when describing the semantics of machine instructions.

3.3 Using τ RTL's to Describe Machines

The previous section described the notation for specifying the effects of single instances of machine instructions. Building on this foundation, we can describe entire instruction sets. Such specifications are commonly called *machine descriptions*.

Given the above syntax, we could describe a machine that is the functional equivalent of a simple calculator by listing each instruction:

$$r[1_{u,2}]_{s,32} \leftarrow r[2_{u,2}]_{s,32} + r[3_{u,2}]_{s,32};$$

$$r[1_{u,2}]_{s,32} \leftarrow r[2_{u,2}]_{s,32} - r[3_{u,2}]_{s,32};$$

$$r[1_{u,2}]_{s,32} \leftarrow r[2_{u,2}]_{s,32} \times r[3_{u,2}]_{s,32};$$

$$r[1_{u,2}]_{s,32} \leftarrow r[2_{u,2}]_{s,32} \div r[3_{u,2}]_{s,32};$$

Such lists are simple enough to build, but lack the descriptive power to concisely specify the set of instructions. In this case, we've shown four different operations that can operate on three different registers. However, this is probably far from complete since each of the three registers can probably be used anywhere. Assuming a machine with four registers, this yields 256 different instances of these four instructions.

We rely, as others have before us, on context-free grammars to describe the language, or instruction set, of the target machine. A grammar for the above example is shown in Figure 3-9. The syntax for the grammars is similar to the those found in texts on formal languages [HU79]. Productions are terminated by the '/' token. Context-free grammars are a good choice for describing sets of instructions since programmers are familiar with Yacc grammars.


```

inst  → reg ← reg op reg; //
reg   → r[numu,2]s,32 //
op    → + | - | × | ÷ //
num   → 1 | 2 | 3 | 4 //

```

Figure 3-9. An τ RTL grammar for a very simple machine.

Unlike string RTLs that use Yacc grammars, τ RTL machine descriptions are not Yacc specifications and are therefore not limited to the set of LALR grammars. τ RTL machine descriptions describe sets of τ RTLs, or sets of trees. So, although there is no LALR restriction, we do require that all machine descriptions describe only valid τ RTL trees. We achieve this by requiring that all grammar nonterminals derive complete τ RTL subtrees. The grammar start symbol (indicated by the first production in the grammar) must derive a complete τ RTL tree.

For example, to extend the grammar of Figure 3-9 to include arbitrary sequences of operations, one might write the grammar shown in Figure 3-10. This grammar describes

```

inst  → reg ← reg op rexpr; //
rexpr → reg op rexpr | reg //
reg   → r[numu,2]s,32 //
op    → + | - | × | ÷ //
num   → 1 | 2 | 3 | 4 //

```

Figure 3-10. An illegal τ RTL grammar.

τ RTL token strings such as:

$$r[1_{u,2}]_{s,32} \leftarrow r[2_{u,2}]_{s,32} - r[3_{u,2}]_{s,32} + r[4_{u,2}]_{s,32};$$

which, at first glance, may appear to be a valid τ RTL. However, since τ RTL does not define an explicit precedence, it is not known if subtraction or addition should be performed first. Further, assuming the subtraction is to be performed first, the type of the result of subtraction has not been specified. The proper specification would be

$$r[1_{u,2}]_{s,32} \leftarrow (r[2_{u,2}]_{s,32} - r[3_{u,2}]_{s,32})_{s,32} + r[4_{u,2}]_{s,32};$$

which specifies the order of evaluation and the result type for the subtraction. Such errors can easily be detected in the τ RTL machine description since the first *rexpr* production in the grammar of Figure 3-10 does not specify a complete τ RTL subtree.

```

inst  → reg ← reg op rexpr, //
rexpr → (reg op rexpr)s,32 | reg //
reg   → r[numu,2]s,32 //
op    → + | - | × | ÷ //
num   → 1 | 2 | 3 | 4 //

```

Figure 3-11. A properly formed τ RTL grammar.

The conciseness of these grammars is due to the use of grammar variables. In a grammar that includes common subexpressions, the same grammar variable may be used. Such identification of common subexpressions is called *grammar factoring* [GHS82]. The introduction of grammar variables has several other benefits. First, names can be given to grammar productions that indicate their purpose (*e.g.*, *inst* obviously derives instructions). Second, where τ RTL notation may be awkward (register index types), an expression can be named and used again and again by name rather than by repeating the awkward expression.

τ RTL grammars make it possible to generalize specific expressions. So concise descriptions can be derived for general machines. However, not all instructions and not all machines are so regular. Take for example, a modified version of our simple machine that only has two-address instructions. On such a machine, an example of an RTL describing a valid instruction would be:

$$r[1_{u,2}]_{s,32} \leftarrow r[1_{u,2}]_{s,32} + r[2_{u,2}]_{s,32};$$

but the RTL:

$$r[1_{u,2}]_{s,32} \leftarrow r[2_{u,2}]_{s,32} + r[3_{u,2}]_{s,32}; \quad (3-14)$$

would not describe a valid instruction because it uses three different registers. We cannot use the grammar rule:

$$inst \rightarrow reg \leftarrow reg \ op \ reg; //$$

since it could derive (3-14). The problem is that both the first and second instance of the grammar variable *reg* are free to derive any of their expressions (any register). τ RTL grammars extend the syntax of context free grammars to solve this common description problem. Grammar variables found on the right-hand side of productions may be tagged with one or more primes (') to name instances of grammar variable derivations. So, the grammar production:

$$inst \rightarrow reg' \leftarrow reg' \ op \ reg; //$$

requires that both `reg`'s must derive the same expression. We limit the scope of matching primed grammar variables to the current production. Grammar variables that are not decorated with primes remain unrestricted in derivations.

There are three additional grammar syntax extensions. τ RTL provides expressions that match any integer constant, any floating-point constant, or any symbolic name (labels, locals, and globals). These expressions are **constant**, **fconstant**, and **name**. So, for a production that derives any register, we would write:

```
reg → r[constantu,5] //
```

rather than having to include an additional production such as:

```
regno → 0 | 1 | 2 | ... | 31 //
```

that defines `regno` as a register number.

Finally, often it is useful to place constraints on the terminals that a grammar variable may derive. On many machines, register zero, when read, always produces zero. To define a grammar that includes all registers, except zero, we would write the production:

```
reg → r[constant'u,5] { constant' ≠ 0 } //
```

the constraint { **constant**' ≠ 0 } must always evaluate to true upon any derivation. The constraint language is limited to C-like Boolean expressions on integer values. Although limited, these expressions provide the necessary power to constrain the derivation of productions in useful ways.

This completes our presentation of τ RTL grammar syntax. Using τ RTL grammars, we can easily define the set of valid instruction semantics for common machines. Figure 3-12 shows a small but complete τ RTL machine description for Hennessy and Patterson's hypothetical DLX machine used for instruction in computer architecture courses [HP96]. Additional machine descriptions can be found in Appendix A.

3.4 Operation Semantics – μ RTL's

The τ RTL notation provides a machine-independent form that uses a set of built-in operations for describing the effects of machine instructions. However, when user-defined operations are used, τ RTL says nothing about the semantics of the user-defined operations. In the presence of user-defined operations, it is only possible, in many cases, to determine what locations have been read and written. In order to provide semantics for operations that are not

1.	<i>start</i>	\rightarrow	<i>inst</i> //
2.	<i>aop</i>	\rightarrow	+ - \times \div //
3.	<i>add_op</i>	\rightarrow	+ - //
4.	<i>mul_op</i>	\rightarrow	\times \div //
5.	<i>bit_op</i>	\rightarrow	\wedge \vee \oplus //
6.	<i>shift_op</i>	\rightarrow	\leftarrow \Rightarrow //
7.	<i>rel_op</i>	\rightarrow	< > \leq \geq //
8.	<i>eq_op</i>	\rightarrow	\equiv \neq //
9.	<i>i</i>	\rightarrow	u s //
10.	<i>int</i>	\rightarrow	u,32 s,32 //
11.	<i>reg</i>	\rightarrow	r[constant _{u,5}] { constant \neq 0 } //
12.	<i>regz</i>	\rightarrow	reg 0 //
13.	<i>freg</i>	\rightarrow	f[constant _{u,5}] //
14.	<i>dreg</i>	\rightarrow	f[constant _{u,5} , 2] //
15.	<i>uregimm</i>	\rightarrow	reg Δ (constant _{u,16}) _{u,32} //
16.	<i>addr</i>	\rightarrow	(reg _{u,32} + Δ (constant _{u,16}) _{u,32}) _{u,32} // <i>addressing modes</i>
17.			regz _{u,32}
18.			Δ (constant _{u,16}) _{u,32} //
19.	<i>jaddr</i>	\rightarrow	PC _{u,32} + Δ (constant _{s,26}) _{u,32} //
20.	<i>jump</i>	\rightarrow	PC _{u,32} \leftarrow <i>jaddr</i> //
21.	<i>link</i>	\rightarrow	r[31 _{u,5}] _{b,32} \leftarrow PC //
22.	<i>inst</i>	\rightarrow	reg _{int'} \leftarrow regz _{int'} <i>add_op</i> regz _{int'} ; <i>arithmetic</i>
23.			reg _{i',32} \leftarrow regz _{i',32} <i>add_op</i> Δ (constant _{i',16}) _{i',32}
24.			freg _{f,32} \leftarrow freg _{f,32} <i>aop</i> freg _{f,32} ;
25.			dreg _{d,64} \leftarrow dreg _{d,64} <i>aop</i> dreg _{d,64} ;
26.			freg _{int'} \leftarrow freg _{int'} <i>mul_op</i> freg _{int'} ;
27.			reg _{b,32} \leftarrow regz _{b,32} <i>bit_op</i> regz _{b,32} ; <i>bitwise operations</i>
28.			reg _{b,32} \leftarrow regz _{b,32} <i>bit_op</i> Δ (constant _{b,16}) _{b,32} ;
29.			reg _{b,32} \leftarrow regz _{b,32} <i>shift_op</i> regz ^{A..0} _{u,5} ; <i>shifts</i>
30.			reg _{s,32} \leftarrow regz _{s,32} \Rightarrow regz ^{A..0} _{u,5} ;
31.			reg _{b,32} \leftarrow regz _{b,32} <i>shift_op</i> constant _{u,5} ;
32.			reg _{s,32} \leftarrow regz _{s,32} \Rightarrow constant _{u,5} ;
33.			reg _{b,32} \leftarrow Δ ((regz _{int'} <i>rel_op</i> regz _{int'}) _{b,1}); <i>compares</i>
34.			reg _{b,32} \leftarrow Δ ((regz _{b,32} <i>eq_op</i> regz _{b,32}) _{b,1});
35.			freg _{s,32} \leftarrow Δ (freg _{f,32}); <i>converts</i>
36.			freg _{f,32} \leftarrow Δ (freg _{s,32});
37.			freg _{d,32} \leftarrow Δ (freg _{f,32});
38.			freg _{f,32} \leftarrow Δ (freg _{d,32});
39.			freg _{s,32} \leftarrow Δ (freg _{d,32});
40.			freg _{d,32} \leftarrow Δ (freg _{s,32});
41.			freg _{b,32} \leftarrow freg; <i>moves</i>
42.			dreg _{b,64} \leftarrow dreg;
43.			freg _{b,32} \leftarrow regz;
44.			reg _{b,32} \leftarrow freg;
45.			reg _{b,32} \leftarrow Δ ((regz _{i',32} <i>rel_op</i> Δ (constant _{i',16}) _{i',32}) _{b,1}); <i>sets</i>
46.			reg _{b,32} \leftarrow Δ ((regz _{b,32} <i>eq_op</i> Δ (constant _{b,16}) _{b,32}) _{b,1});
47.			FCC _{b,1} \leftarrow freg _{f,32} <i>rel_op</i> freg _{f,32} ;
48.			FCC _{b,1} \leftarrow freg _{b,32} <i>eq_op</i> freg _{b,32} ;
49.			FCC _{b,1} \leftarrow dreg _{d,64} <i>rel_op</i> dreg _{d,64} ;
50.			FCC _{b,1} \leftarrow dreg _{b,64} <i>eq_op</i> dreg _{b,64} ;

Figure 3-12. A complete τ RTL machine description of the DLX .

51.		$reg_{b,32} \leftarrow \Delta(\mathbf{constant}_{b,16})_{b,32} \leftarrow 16_{u,5};$	<i>loads</i>
52.		$reg_{i,32} \leftarrow \Delta(m[addr]_{i,8});$	
53.		$reg_{i,32} \leftarrow \Delta(m[addr, 2]_{i,16});$	
54.		$reg_{b,32} \leftarrow m[addr, 4];$	
55.		$freg_{b,32} \leftarrow m[addr, 4];$	
56.		$dreg_{b,64} \leftarrow m[addr, 8];$	
57.		$m[addr]_{b,8} \leftarrow regz^{7..0};$	<i>stores</i>
58.		$m[addr, 2]_{b,16} \leftarrow regz^{15..0};$	
59.		$m[addr, 4]_{b,32} \leftarrow regz;$	
60.		$m[addr, 4]_{b,32} \leftarrow freg;$	
61.		$m[addr, 8]_{b,64} \leftarrow dreg;$	
62.		<i>jump;</i>	<i>jumps</i>
63.		<i>jump; link;</i>	
64.		$PC_{b,32} \leftarrow regz;$	
65.		$PC_{b,32} \leftarrow regz; link;$	
66.		$PC_{u,32} \leftarrow ?((FCC_{b,1} \equiv 1_{b,1})_{b,1}, jaddr, PC_{u,32});$	<i>branches</i>
67.		$PC_{u,32} \leftarrow ?((FCC_{b,1} \equiv 0_{b,1})_{b,1}, jaddr, PC_{u,32});$	
68.		$PC_{u,32} \leftarrow ?((regz_{b,32} eq_op 0_{b,32})_{b,1}, jaddr, PC_{u,32});$	
69.		$PC_{u,32} \leftarrow trap(\mathbf{constant}_{u,26});$	
70.		$PC_{u,32} \leftarrow rfe();$	
		//	

Figure 3-12. A complete τ RTL machine description of the DLX (*Continued*).

built-in (all built-in operations are listed in Table 3-4), we rely on an operational semantics using a language called μ RTL. μ RTL is designed to describe instructions at the micro-architecture level. Details that are not of use at the τ RTL level can be provided in a μ RTL description.

The concept of μ RTL is simple: describe the semantics of user-defined operations using the built-in operations. Once one understands the τ RTL notation, one can easily begin specifying the semantics of user-defined operations. For example, a good candidate for a user-defined operator is string copy. String copy copies an array of bytes from one location to another. The array is terminated by a byte whose value is zero. This instruction cannot be described completely at the τ RTL level because it contains an internal loop. Further, the details of the semantics are really too complicated to be present at the τ RTL level. Instead, we can rely on μ RTL to describe string copy's semantics. A τ RTL that uses the user-defined operator `strcpy` might look like:

```
 $\leftarrow \text{strcpy}(reg_{u,32}, reg_{u,32});$ 
```

Category	Operations
Signed arithmetic $s, n \times s, n \rightarrow s, n$	$+$, $-$, \div , mod (modulus), $\times_{s, n \times s, n \rightarrow s, 2n}$, $-_{s, n \rightarrow s, n}$ (unary minus)
Unsigned arithmetic $u, n \times u, n \rightarrow u, n$	$+$, $-$, \div , mod (modulus), $\times_{u, n \times u, n \rightarrow u, 2n}$
Floating-point arithmetic $f, n \times f, n \rightarrow f, n$	$+$, $-$, \div , \times
Signed relational $s, n \times s, n \rightarrow b, 1$	$<$, $>$, \leq , \geq
Unsigned relational $u, n \times u, n \rightarrow b, 1$	$<$, $>$, \leq , \geq
Equality $b, n \times b, n \rightarrow b, 1$	\equiv , \neq
Bitwise $b, n \times b, n \rightarrow b, n$	\wedge , \vee , \oplus , \neg (complement)
Logical $b, 1 \times b, 1 \rightarrow b, 1$	\wedge , \vee , \oplus , \neg (not)
Bitwise shift $b, n \times u, m \rightarrow b, n$	\leftarrow , \Rightarrow
Signed shift $s, n \times u, m \rightarrow s, n$	\leftarrow , \Rightarrow
Type Conversion	$\Delta_{s, n \rightarrow u, n}$ $\Delta_{u, n \rightarrow s, n}$ $\Delta_{s, n \rightarrow s, m}$ (where $n < m$) (sign extend) $\Delta_{u, n \rightarrow u, m}$ (where $n < m$) $\Delta_{f, n \rightarrow f, m}$ $\Delta_{s, n \rightarrow f, m}$ $\Delta_{f, n \rightarrow s, m}$
Selection $b, 1 \times b, n \times b, n \rightarrow b, n$?

Table 3-4. Summary of τ RTL built-in operations.

where both *reg*'s specify the starting memory addresses for the destination and source arrays. The lack of a storage location on the left hand side of ' \leftarrow ' indicates that the operation does not store a value (beyond the setting of bytes pointed to by the destination register).

To specify *strcpy*'s semantics, we simply write a sequence of τ RTL built-in operations that implement the operation. Such an implementation is shown in Figure 3-13. On line 1, we specify the name of the operation, the number and type of operands, and the type of the

result (in this case none). Local variables and their sizes (in bits) are declared on line 2. Such local variables might correspond to hidden registers in the machine's microarchitecture. Because string copy has an internal loop, we label line 5 (the top of the loop) with a μ RTL label. Jumps are accomplished by assigning to the μ RTL program counter (μ PC). In this case, we want a conditional branch at the bottom of the loop (line 10), so we use the μ RTL '?' operator (analogous to the '?' operator in C) to select between loop and the current value of μ PC based on the value of the Boolean variable t4. So, the only additions to the τ RTL language are the operation's header, a syntax for declaring temporary storage locations, labels, and the μ RTL program counter.

```

1.  ← strcpy(destu,32, sourceu,32) {
2.      var t1:32, t2:32, t3:8, t4:1;
3.      t1b,32 ← dest;
4.      t2b,32 ← source;
5.  loop: t3b,8 ← m[t2u,32];
6.      m[t1u,32]b,8 ← t3;
7.      t1u,32 ← t1u,32 + 1u,32;
8.      t2u,32 ← t2u,32 + 1u,32;
9.      t4b,1 ← t3b,8 ≡ 0b,8;
10.     μPCb,32 ← ?(t4b,1, loopb,32, μPCb,32);
11. }
```

Figure 3-13. μ RTL operational semantics for a user-defined string copy operator.

Figure 3-14 illustrates another use of μ RTL to specify the semantics of the Pentium PADDB instruction [Int93]. PADDB performs a packed-addition on bytes. PADDB adds the individual bytes of a 64-bit source operand to the individual bytes of a 64-bit destination operand. A carry out of an individual byte addition is not propagated into the adjacent byte. We can describe PADDB by introducing a new type 'p' (packed) using the τ RTL¹:

$$dest'_{p,64} \leftarrow \text{paddb}(dest'_{p,64}, src_{p,64});$$

The μ RTL implementation uses a loop to add each of the individual quantities separately. Thus, μ RTL is an ideal mechanism for specifying microinstruction-level detail without cluttering the τ RTL notation.

1. We have used the τ RTL grammar syntax to illustrate that the destination operand must correspond to one of the source operands. We also use this form to abstract away unnecessary details of the source and destination address calculations.

```

1.  destp,64 ← paddb(src1p,64, src2p,64) {
2.      var lo:8, hi:8;
3.      lou,8 ← 0;
4.      hiu,8 ← 7;
5.  loop:  desthi..lou,8 ← src1hi..lou,8 + src2hi..lou,8;
6.      hiu,8 ← hiu,8 + 8u,8;
7.      lou,8 ← lou,8 + 8u,8;
8.      μPCb,32 ← ?((hiu,8 < 64u,8)b,1, loopb,32, μPCb,32);
9.  }
```

Figure 3-14. Operational semantics for the Pentium PADDB instruction.

3.5 Summary

In this chapter, we have presented a new formulation of RTLs that address shortcomings of string RTLs. Using τ RTLs, the semantics of instructions are presented in both a machine independent and application-independent manner. As with string RTLs the abstractions that τ RTLs provide are well suited for many systems software applications. Additionally, τ RTL types match the types found at the assembly language abstraction level by associating the types with the operations rather than with operands as high-level languages do. This prevents the ambiguities found in string RTLs and increases the readability and processability of the τ RTL form.

Although the RTL level of abstraction has often been demonstrated to be appropriate for systems software, there are times when too many details are abstracted away. To accommodate the need for additional detail, without cluttering τ RTLs, we have introduced a new method for describing both the semantics of user-defined τ RTLs and features of a machine's microarchitecture: the μ RTL. By using this hierarchical approach, τ RTL in combination with μ RTL can satisfy the needs of a wider audience of applications than previous description systems.

Finally, we have shown how grammars for τ RTLs can be constructed to describe the language that corresponds to a machine's instruction set. Unlike string RTL descriptions, and most other description systems, these descriptions maintain the application-independent nature of τ RTLs. In fact, in this chapter, these descriptions stand alone, specifying the language of τ RTLs that corresponds to a machine's capabilities without specifying how such information might be used. As stand-alone, τ RTL grammars can be used to recognize a machine's capabilities by using the description's grammar as a τ RTL recognizer in the tradi-

tional way. Further, the grammar can be used to enumerate all valid τ RTLs for the target machine. Although isolated from purpose, recognition is used in code optimization while enumeration has uses in test generation. Nonetheless, in Chapter 5, we will show how these specifications can be extended to include additional types of information—both application-specific and application-independent.

CHAPTER 4

SPECIFYING PROCEDURE CALLING CONVENTIONS

Beyond the core CSDL module, additional modules can be defined that describe features of computing systems not covered by the core description. In this chapter, we examine a module that describes procedure calling conventions. The *procedure calling convention* dictates the way that program values are communicated across procedure calls, and how machine resources are shared between a procedure making a call (the *caller*) and the procedure being called (the *callee*). The procedure calling convention represents one of CSDL's highest levels of abstraction because it spans the software-hardware boundary.

In this chapter, we introduce two important new concepts: 1) the distinction between the procedure calling convention and its implementation (the calling sequence), and 2) the recognition that two procedures create two distinct points of view by naming the same memory locations with different names. These concepts, in themselves are important. However, by using them, we are able to build both more robust implementations and identify and isolate faults in less robust implementations. Finally, we also demonstrate how, by building application-independent descriptions, we can use the same description for multiple purposes.

4.1 Introduction

The procedure calling convention impacts the operation of many systems software components. The interface between procedures, which is established by the calling convention, facilitates separate compilation of program modules and interoperability of programming languages. What makes calling conventions unique and interesting is that they are not imple-

mentation dependent or entirely language dependent. Instead, the calling convention is machine-dependent because the rules for passing values from one procedure to another depend on machine-specific features such as memory alignment restrictions and register usage conventions. Further, code that implements the calling convention must be generated by the compiler and understood by other systems software.

4.1.1 Motivation

Currently, information about a particular calling convention can be found by looking in the programmer's reference manual for the given machine, or reverse-engineering the code generated by one of its compilers. Reverse-engineering a compiler has many obvious shortcomings. Using the programmer's reference manual may be equally problematical. As with much of the information in the programmer's manual, the description is likely to be written in English and is liable to be ambiguous, or inaccurate. For example, in the MIPS programmer's manual [KH92] the English description is so difficult to understand that the authors provide fifteen examples, several of which are contradictory [Fra93]—and this is the *second* edition of the programmer's manual. Furthermore, the convention, once understood, is difficult to implement. For example, the GNU ANSI C compiler fails on an example listed in the manual. Digital Equipment Corporation, in recognizing the problem, has published a calling standard document for their Alpha series processors [Dig93] that exceeds 100 pages¹. Thus, it should be clear that there is a need for accurate, concise descriptions of procedure calling conventions and software to use them.

One reason that such a need exists is that the few investigations of calling sequences have been *ad-hoc*. For example, Johnson and Richie discuss some rules of thumb for designing and implementing a calling sequence for the C programming language [JR]. Davidson and Whalley experimentally evaluated several different C calling conventions [DW91]. However, no work has been done to formally analyze calling conventions.

1. Although this document also includes information on exception handling and information pertinent to multithreaded execution environments, more than 42 pages are devoted to documenting the calling convention.

4.1.2 Applications

Any application that must process, or generate procedures at the machine-language abstraction level is likely to need to know about a procedure calling convention. Example applications include compilers, debuggers, linkers and evaluation tools such as profilers. The code that implements the calling convention in these applications lends itself to automatic generation. Often, the convention itself is not difficult to understand or implement, for a given instance of a procedure call. However, a general solution that covers all possible cases is difficult to implement correctly.

To address these problems, we have developed a formal specification language for describing procedure calling conventions. This language, called *CCL* (Calling Convention Language), has been used to generate automatically the calling sequence generator for a compiler [BD95]. The compiler, called *vpcc/vpo*, is a retargetable optimizing compiler for the C language that has been targeted to over a dozen different architectures [BD88, BD94].

The procedure calling convention for a target machine is described using CCL. The resulting specification is processed by an interpreter that generates tables that can be used in the calling-convention-specific portion of *vpcc/vpo*, or in a test suite generator. Figure 4-1 shows this process. The test suite generator uses information from the table to build a test suite for the specific calling convention. The test suite can be used to either confirm that the *vpcc/vpo* implementation properly uses the convention tables, or confirm that another, independent compiler conforms to the convention described in the CCL specification.

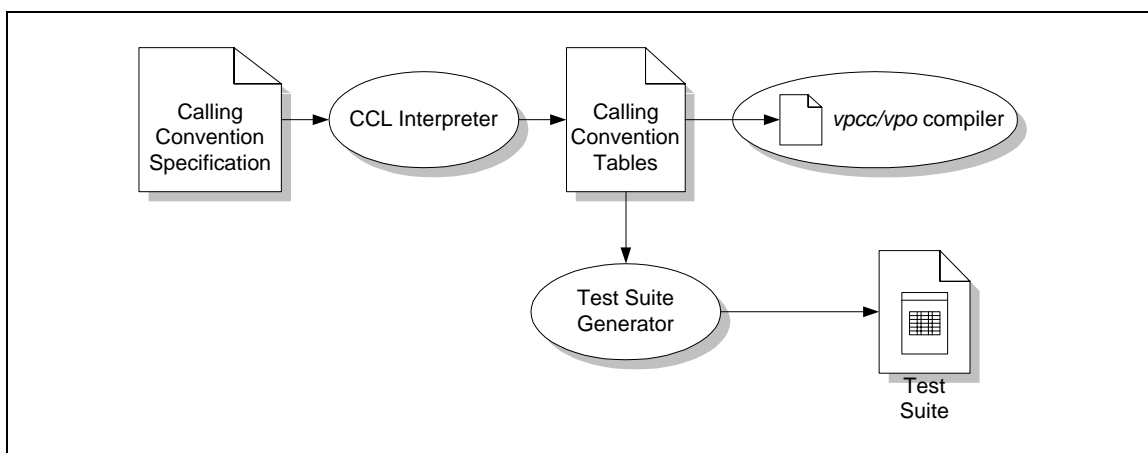


Figure 4-1. How CCL specifications are used.

4.2 Procedure Calling Conventions

To facilitate local compilation of procedures, compiler developers establish rules about how procedures interact. These rules establish an agreement between the caller and callee on how information and control are passed between the two, as well as how and who will maintain the state of the machine. Collectively, these rules are known as the procedure calling convention.

4.2.1 A Simple Calling Convention

To aid in our discussion of calling conventions, we use a simplified example calling convention. Figure 4-2 contains the calling convention rules for a hypothetical machine. Consider the following ANSI C prototype for a function `warp`:

```
int warp(char p1, int p2, int p3, double p4);
```

For the purpose of transmitting procedure arguments for our simple convention, we are only interested in the *signature* of the procedure. We define a procedure's signature to be the procedure's name, the order and types of its arguments, and its return type. This is analogous to ANSI C's *abstract declarator* [KR88], which for the above function prototype is:

```
int warp(char, int, int, double);
```

which defines a function that takes four arguments (a char, two int's, and a double), and returns an int.

1. Registers a^1 , a^2 , a^3 , and a^4 are 32-bit argument-transmitting registers.
2. Arguments are also passed on the stack in increasing memory locations starting at the stack pointer ($M[sp]$).
3. An argument may have type char (1 byte), int (4 bytes), or double (8 bytes).
4. An argument is passed in registers (if enough are available to hold the entire argument), and then on the stack.
5. Arguments of type int are 4-byte aligned on the stack.
6. Arguments of type double are 8-byte aligned on the stack.
7. Stack elements that are skipped over cannot be allocated later.
8. Return values are passed in registers a^1 and a^2 .
9. Values of registers a^6 , a^7 , a^8 , and a^9 must be preserved across a procedure call.

Figure 4-2. Rules for a simple calling convention.

With `warp`'s signature, we can apply the calling convention in Figure 4-2 to determine how to call `warp`. Arguments to `warp` would be placed in the following locations:

- `p1` in register `a1`,
- `p2` in register `a2`,
- `p3` in register `a3`, and
- `p4` on the stack in `M[sp:sp + 7]` (`M` denotes memory).

Notice that although register `a4` is available, `p4` is placed on the stack since it cannot be placed completely in argument-transmitting registers (rule 4). Such restrictions are common in actual calling conventions.

4.2.2 Convention, Language, and Implementation

The first thing to notice about our simple calling convention is the lack of detail. There are many questions that are left unanswered. Among them are:

- In what order are the procedure's arguments evaluated?
- In what order are the procedure's arguments placed in registers and on the stack?
- Where are the persistent¹ registers stored?
- Which persistent registers need to be saved?
- What is the activation frame layout?

Each of these questions must be answered in order to produce a working implementation.

These questions are answered by two other elements that interact with the procedure calling convention: the definition of the procedure's source language and the language's implementation. In this work, we have made a conscious effort to separate the concepts of calling convention, language definition, and implementation.

The choice to isolate the concepts of the convention from those of the language definition is an obvious one. To facilitate inter-language procedure calls, a single convention separate from the language definition, must be available. There are, however, features of the source language that may be present in the convention. For example, in our hypothetical convention, *where* an argument is placed is determined, in part, by the type of the argument. Such language features cannot be avoided in the description of the convention, but they should be

1. A register whose value is preserved across a procedure call is *persistent*.

kept to a minimum. Also, it illustrates what features both languages must share to make inter-language procedure calls possible at all.

The need for the second separation, between the convention and the language implementation, may be less obvious. Compiler writers commonly refer to the mechanism by which procedure calls are made as either the calling convention, or the calling sequence. Although these two terms are frequently used interchangeably, they are separate concepts and we treat them as such. Without additional information, the calling convention itself does not provide enough information to produce an implementation. The calling sequence, on the other hand, is an implementation of the calling convention. It is a sequence of machine instructions that implement a procedure call. There may be many calling sequences for a given calling convention. Furthermore, since the sequence implements the convention, the caller cannot distinguish between two different calling sequences used by the callee, and *vice versa*. Thus, while it is imperative that a caller and a callee use the same calling convention, it is not necessary that they use the same calling sequence.

4.2.3 Separating Convention from Sequence

An important result of this work is the identification of calling convention and calling sequence as separate concepts. Although at first this distinction may seem unnatural, it has many benefits. The reason it seems unnatural is that the two concepts are so closely coupled. It is impossible to discuss calling sequences without calling conventions. However, the reverse is not true. By extracting the concept of convention from the calling sequence, we are able to more accurately model the interaction between procedures and the interaction between systems software that process procedures.

When discussing calling conventions, we have found it useful to have a litmus test that helps us identify what features of the procedure call are part of the calling convention, and what features are part of the calling sequence. We ask the following question:

If I change the implementation of this feature on one side of the procedure call, will it impact the other side of the call?

If the answer to this question is “yes,” then the feature is part of the calling convention. If “no,” the feature is part of the calling sequence. For example, if the callee changes where it

stores the values of persistent registers that it uses, the caller need not be changed. Thus, where these values are stored is a feature of the calling sequence. Conversely, if the callee changes where it stores its return value, the caller must also be changed so it can properly retrieve the value upon return. Therefore, the placement of the return value is a matter of calling convention.

Separating the convention from the sequence is often quite difficult. Conventions are usually illustrated using sequences, and when considering a convention, it is natural to think of how it might be implemented—the sequence. However unnatural this process may at first seem, we have reaped great benefits by performing this delicate separation. The calling convention descriptions that we have developed are more accurate, and can be used for many applications primarily due to their application independence. A good deal of this independence is due to the specification of convention without consideration of sequence. The result is specifications that describe how procedures must interact at a high level without describing the implementation of these interactions. Descriptions of such implementations would have many obvious shortcomings.

4.2.4 Interfaces and Agents

So far, we have mentioned a single procedure call interface. Actually, there are two interfaces: the procedure call interface and the procedure return interface as shown in Figure 4-3. We model the actions and responsibilities on each side of these interfaces using agents. An *agent* ensures that its side of the interface satisfies the requirements of the calling convention. These agents are the *whom* in the definition of the calling convention. For the procedure call interface, there are the *caller prologue* and *callee prologue* agents that are responsible for correctly passing the procedure arguments and constructing an environment in which the callee can execute. For the procedure return interface, there are the *callee epilogue* and *caller epilogue* that are responsible for correctly passing the procedure return values and restoring the environment of the caller. The responsibilities of each of the four agents are closely related. The caller prologue and callee prologue agents must agree on how to pass information, as must the caller epilogue and callee epilogue. Additionally, actions of the epilogue agents must be symmetric to the actions of the prologue agents to properly restore the environment (*e.g.*, if the call dec-

rements the stack pointer, the return must increment it). It is precisely these restrictions that make it difficult to correctly construct a calling sequence.

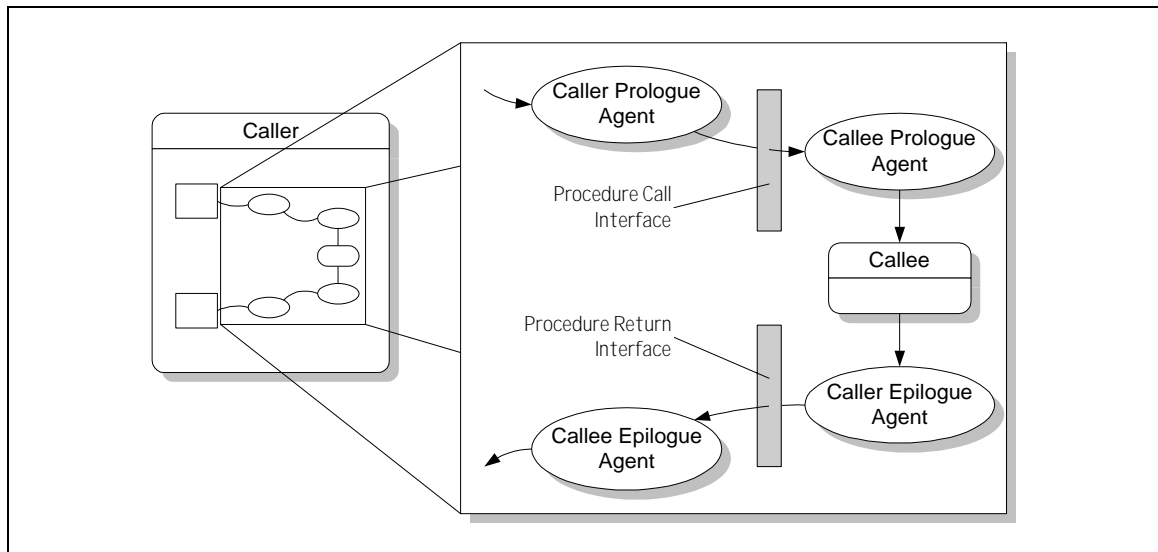


Figure 4-3. The role of agents in procedure call and return interfaces.

4.2.5 Addressing

One responsibility of each agent is to maintain the environment in which procedures execute. Depending on the language and its implementation, the environment can contain arbitrary information. However, one aspect of the environment that almost all languages are likely to share is the concept of addressing. Addressing describes how a name in the source language is bound to a location in the implementation. For example, local variables are commonly found on the stack, while global variables may be referenced through a global space pointer.

Sometimes, to properly construct an environment for a procedure, the caller must provide to the callee details about the caller's environment. For example, in Pascal [KJ74], where nested procedures can refer to variables in the scope of their containing procedures (up-level references), the caller must provide the callee with environment information for the callee to properly implement the scoping rules of the language. Using our litmus test, clearly the structure of the environment information is part of the calling convention. If the structure were changed, the callee would need to be changed so it could properly find variables that are visible to the callee.

Although the structure of information that is transmitted between procedures is a matter of convention, we have not included it in our convention specifications. Just as it is

reasonable to discuss calling convention rules using data types that are never formally defined, it is also reasonable to specify how information is passed between procedures without defining its structure. We believe that description of the structure of information is itself an interesting and difficult problem that is best left as a future research effort.

4.2.6 Activation Frame Layout

An important decision that must be made when implementing a procedure calling convention is the layout of the procedure activation frame. An activation frame is one of several implementation choices for storing the information specific to a particular activation of a procedure. A surprising result of studying calling conventions is that a complete specification of the calling convention is unlikely to determine the frame layout.

Information that is typically found in a procedure activation frame includes the procedure's parameters, locations for storing local variables and temporaries, space for saving the values of persistent registers, and space for any other environment information. Where this information is found in the frame is determined, in part, by the convention and, in part, by the implementation. The convention fixes the location of the procedure's arguments, while it is up to the implementation to specify where local variables are stored. Thus, any implementation must make some decisions about frame layout. Section 4.5.2 discusses how this is done in our implementation.

4.3 The CCL Specification Language

In this section, we present the specification language that we use to describe procedure calling conventions. Once a convention is specified in CCL, we avoid the pitfalls related to using the programmer's reference manual and reverse-engineering the compiler. We present the key features of CCL and enough syntax of the language to understand the examples.

4.3.1 Design Philosophy

In designing CCL, there were a number of features that we wanted to be include. First and foremost, the language had to be processed automatically. Second, we wanted the descriptions to be natural. Hence, the elements of the language had to reflect concepts common to calling conventions. Third, the design had to avoid over-specification of conventions. To achieve this, we tried to exploit the symmetry of the procedure call to eliminate redundancy in the descrip-

tions. Finally, the feature that received the least priority was the syntax of the language. The selection of what symbols to use for operators, for example, was only of secondary concern.

We describe the key features of CCL by presenting a simple CCL description. We develop the example description piece-by-piece as each part of description is presented. This section concludes with a complete description of our hypothetical calling convention in Figure 4-5. As with other modules in CSDL, CCL uses an extended ASCII character set and typographical elements such as bold face, superscripts and special fonts.

4.3.2 Resources

The primary objects in CCL are resources. A *resource* is simply any location that can store a value. Examples include registers and memory locations, such as the stack. Since resources correspond to a machine's memory locations, CCL models them as arrays. The name of the resource is bold and indices are usually superscripted. For example, \mathbf{a}^5 would designate register five. Memory indices are treated similarly: \mathbf{M}^{1023} to designate address 1023. However, the combination of the two—such as using a register to compute a memory address—would yield expressions such as: $M^{\mathbf{a}^5 + 12}$. Therefore, we also permit indices to be specified within brackets ('[' and ']') to avoid superscripting superscripts: \mathbf{M}^{1023} may be written as $\mathbf{M}[1023]$. This treatment is identical to the mathematics convention of writing e^x as $\exp(x)$ when x is typographically complicated.

In addition to resources, CCL descriptions use sets. Sets contain elements that are either integers, resources, or sets. Where possible, CCL uses standard mathematical notation for building sets. For example, the expression:

$$\{1, 2, 3, 4\}$$

describes the set containing the first four positive integers. It is most useful to define sets of resources, such as the first four locations of the stack (if \mathbf{a}^5 is the stack pointer):

$$\{\mathbf{M}[\mathbf{a}^5], \mathbf{M}[\mathbf{a}^5 + 1], \mathbf{M}[\mathbf{a}^5 + 2], \mathbf{M}[\mathbf{a}^5 + 3]\}$$

such expressions can be more concisely expressed by using the range operator (':'). An equivalent formulation of the resource set above is: $\{\mathbf{M}[\mathbf{a}^5:\mathbf{a}^5 + 3]\}$. Infinite sets can be constructed with the use of the special integer infinity ('∞'). To describe the set of all memory locations

addressable from the stack pointer, use $\{M[sp:\infty]\}$ where ‘sp’ is an alias for the stack pointer. Such an alias can be introduced using the **alias** statement:

$$\mathbf{alias\ sp\ \equiv\ a^5}$$

More complicated sets can be constructed using the condition operator (‘|’) that usually is accompanied by the introduction of a set variable. For example, the expression:

$$\{ M[addr] \mid addr \underline{\text{mod}}\ 4 = 0 \}$$

describes all of the four-byte-aligned memory locations. In this expression the variable *addr* is introduced in the set expression and is then constrained within the condition expression.

Obviously, set variables are indicated by using italics. Such conditions may include the standard relational operators and set membership (‘∈’) bound together using Boolean AND (‘^’). Finally, in addition to sets, CCL allows for ordered sets (often called lists) by changing the delimiters to angle brackets (‘<’ and ‘>’): $\langle M[sp:\infty] \rangle$. Ordered sets define the order in which elements can be extracted or considered.

CCL descriptions are composed of five sections: a global declaration section and a section for each of the four agents (caller prologue, callee prologue, callee epilogue and caller epilogue). The global section introduces names that are used in two or more agent descriptions. An agent description, if present, may include zero or more resource placements and zero or more view changes. We describe each of these in detail in the following paragraphs.

4.3.3 Global Section

The global section introduces names and defines properties that impact all convention agents. Three types of statements may be used in the global section: **external**, **persistent**, and **alias**. An example global section is:

$$\begin{aligned} &\mathbf{external\ NVSIZE,\ SPILL_SIZE,\ LOCALS_SIZE} \\ &\mathbf{persistent\ \{a^6,\ a^7,\ a^8,\ a^9\}} \\ &\mathbf{alias\ sp\ \equiv\ a^5} \end{aligned}$$

The **external** statement indicates identifiers that whose values are defined by the outer environment (discussed below). The **persistent** statement identifies those machine resources whose values must persist¹ (be preserved) across a procedure call. Finally, the **alias** statement, which we already have seen, introduces more meaningful names for expressions.

1. Persistent resources are often called *callee save* locations, or *non-volatile locations*.

4.3.3.1 Outer Environment

Although CCL is used to capture information about a calling convention, a CCL description does not contain all necessary information to produce a calling sequence. Indeed, CCL descriptions are not complete by themselves. CCL descriptions require information from the outer CSDL environment to complete the descriptions. Information about the machine and language, such as the size of registers, the base data types and local procedure information, such as the amount of space needed for temporary variables, and which registers are used, are provided by other components of the CSDL description system. Four variables that are always defined by the outer environment are the special resources **ARG**, **RVAL**, and the corresponding special resource sizes 'ARG_TOTAL' and 'RVAL_TOTAL'. Since these values are always defined, they are implicitly declared as external values. All other variables whose values are provided by the outer environment are declared using the **external** statement.

4.3.4 Agent Descriptions

The responsibilities agents are specified in each agent's corresponding section. An agent's responsibilities fall into one of two categories: data transfers, and view changes. Thus, each agent's section may be composed of zero or more data transfer and view change statements. Data transfers describe movement of values from one resource to another that the agent must perform, while view changes describe shifts in an agent's point-of-view with regard to machine resources.

4.3.4.1 Data Transfers

The primary responsibility of each agent is to make possible the transmission of information from the caller to the callee and *vice versa*. These transmissions are specified using a **data transfer** statement within an agent's section. The **data transfer** statement always includes a **resources** declaration and a **map** statement. In addition, new aliases, constants, and names may be introduced in the data transfer statement.

Each parameter or return value must be assigned a resource to communicate the value across the call. These locations are taken from the set of declared mapping resources. In declaring these resources, two things must be considered: what resources may be allocated for data transmission, and in what order may they be allocated. The resources declaration specifies both of these. For example, the statement:

resources {<a^{1:4}, M[sp:∞]>}

indicates that the first four ‘a’ registers and all stack memory locations may be targets for moves. The inner set is ordered because once a resource later in the list is used (such as a⁴), resources earlier in the list may no longer be considered (such as a²).

Once a set of target resources has been declared, the resources are partitioned into classes. A *class* is an ordered set of ordered sets that indicate where to *start* placing a value. For example, we could introduce the ‘regs’ class using the following **class** statement:

class regs ← <<register> | register ∈ <a^{1:4}>>

This binds the ‘regs’ identifier to an ordered set of ordered sets, each of which contains a single register. This set is equivalent to:

class regs ← <<a¹>, <a²>, <a³>, <a⁴>>

The inner ordered set is used to indicate a list of starting locations if a value cannot be placed in a single resource.

Once the set of resources has been declared and partitioned into classes, the actual mapping of values to resources can be specified using the **map** statement. In its simplest form, **map** takes a source resource and maps it to a class. For example, the map statement:

map ARG¹ → <regs, mem>

takes a single resource as the source (left hand side) and an ordered set classes as the destination. In this case, ARG¹ would be mapped to a location in the ‘regs’ class, or if none were available, ARG¹ would be mapped to a location in the ‘mem’ class.

During the mapping process several resource attributes are examined and set. A resource *attribute* is specified by naming the resource followed by a period, followed by the name of the attribute. There are four resource attributes in CCL:

- ‘type’: the type of the parameter or return value (defined for ARG and RVAL resources only).
 - ‘size’: the size of the resource in bytes.
 - ‘assigned’: a Boolean value, that if true, indicates the resource has been used in a mapping.
 - ‘unavailable’: a Boolean value, that if true, indicates the resource may not be used in a mapping.
-

The map operator uses the ‘size’, ‘assigned’, and ‘unavailable’ attributes. The ‘type’ attribute is often used to select one of many possible targets for a mapping. For example, in combination with the case expression (\perp), the lines:

```

map ARG1 → ARG1.type  $\perp$  {
  char:      <regs, mem>,
  int:       <regs, imem>,
  double:    <regs, dmem>,
}

```

select one of three sets of ordered sets based on the value of ARG¹'s type and maps ARG¹ to the set.

4.3.4.2 Conditionals and Iteration

As with many languages, CCL has expressions for both conditional evaluation and iteration. Both types of expressions introduce variables whose scope is the enclosed expression. The universal quantifier (\forall) operator iterates over a set, each time binding the variable to an element of the set. For example, to perform mappings for all arguments, we could use iteration in combination with **map**:

```

 $\forall$  argument  $\in$  <ARG1:ARG_TOTAL>
  map argument → <regs, mem>

```

In this example, the variable *argument* is bound to each element of the set in order (the set is ordered), and the statement is evaluated with that binding. Similarly, the existential quantifier (\exists) can be used to perform a statement conditionally.

4.3.4.3 Internal Values

The final type of agent statement is the **internal** statement. In contrast to external, internal values are defined by the CCL description. The **internal** keyword is used to introduce a named integer value. Most often, it is used to compute sizes for various parts of an activation frame. In our example, we define ‘ARG_SIZE’ using:

```

internal ARG_SIZE  $\leftarrow$   $\Sigma$ (<M[addr].size | addr  $\in$  <mindex>  $\wedge$  M[addr].assigned>)

```

Expressions on the right-hand-side of ‘ \leftarrow ’ may be simple integer expressions or the special summation operator (Σ) used here that sums the values of a set of integers. In this example, we compute the sum of the sizes of each memory resource whose assigned attribute is set. This corresponds to the amount of memory that the above map operator used when placing the arguments.

At this point, we coalesce our CCL statements presented in this section to yield the entire data transfer block for the caller prologue of our simple convention. The specification is shown in Figure 4-4. To review, the two aliases ‘mindex’ and ‘argregs’ are introduced to name the possible argument memory indices and argument registers, respectively. The set of destinations for placement specified as the registers followed by memory in the **resources** statement. Three subsets of these resources are defined as ‘regs’, ‘imem’, and ‘dmem’ using the **class** statement. The ‘regs’ class contains all argument registers, ‘imem’ contains all four-byte-aligned memory locations, and ‘dmem’ contains all eight-byte-aligned memory locations. The **map** statement, in combination with iteration and the case selection operator, maps all arguments to registers and various memory locations based on the type of the argument. In all cases, registers are considered first, and then memory. When registers are exhausted, ints are placed in four-byte-aligned memory while doubles are aligned by eight bytes. Finally, the amount of memory used by map is computed and placed in the variable ‘ARG_SIZE’ for use in other parts of the description.

```

1.      alias mindex  $\equiv$  sp: $\infty$ 
2.      alias argregs  $\equiv$  a1:4
3.      resources {<argregs, M[mindex]>}
4.      class regs  $\leftarrow$  <<register> | register  $\in$  <argregs>>
5.      class imem  $\leftarrow$  <<M[addr]> | addr  $\in$  <mindex>  $\wedge$  addr mod 4 = 0>
6.      class dmem  $\leftarrow$  <<M[addr]> | addr  $\in$  <mindex>  $\wedge$  addr mod 8 = 0>
7.       $\forall$  argument  $\in$  <ARG1:ARG_TOTAL>
8.          map argument  $\rightarrow$  argument.type  $\perp$  {
9.              char:      <regs, M[mindex]>,
10.             int:       <regs, imem>,
11.             double:    <regs, dmem>,
12.          }
13.      internal ARG_SIZE  $\leftarrow$   $\sum$ (<M[addr].size | addr  $\in$  <mindex>  $\wedge$ 
14.          M[addr].assigned>)

```

Figure 4-4. The caller prologue.

4.3.4.4 View Changes

In addition to data transfers, CCL can describe changes in an agent’s point of view. Such changes, called *view changes*, describe shifts in resource names and are an important CCL innovation. Without view changes, we cannot describe the effects of procedure memory allocation such as stack allocation and register windows.

When a procedure is called, memory must be allocated somewhere to store the procedure's arguments and local variables. When a procedure returns, this memory must be deallocated. This space may consist of registers or memory, or a combination of the two. Often, this allocation takes the form of stack allocation. Since stack allocation requires a change in the stack pointer, an unfortunate side-effect occurs: all memory locations that are referenced through the stack pointer change names. For example, say a local variable X is stored at $M[sp + 12]$. If, in the process of allocating an activation frame, the stack pointer is decremented by 48, the name for X must change to $M[sp + 60]$ to reflect the shift in the value of 'sp'. Such shifts happen in all calling conventions, but are often difficult to express.

In CCL, we could model shifts in names using a data transfer statement. However, this would imply that the values actually moved and would cause moves to occur. Instead, CCL allows such shifts to be expressed using view changes that, in turn, are mapped to actions that cause such shifts to occur without moving the data. For example, in our simple convention, allocation is performed using the stack. Within a **view change** statement, the **becomes** keyword can be used to express this shift for a single location:

$$M[sp + 12] \text{ becomes } M[sp + 60]$$

When many such changes in view occur, iteration can be used:

$$\forall \text{ offset} \in \{-\infty; \infty\} \\ M[sp + \text{offset}] \text{ becomes } M[sp + \text{offset} + 48]$$

This corresponds to a push of a 48-byte activation frame onto the stack.

The view change concept encompasses more than just stack allocation. Any action that causes names to change can be described using changes in view. The register window mechanism on the SPARC microprocessor is an example. When the register window slides, the contents of the registers appear to move because the names of the registers have changed. This shift can be expressed in exactly the same way that stack pushes and pops are expressed.

4.3.4.5 Symmetry

The data transfer and view change statements are sufficient to describe the agent responsibilities of complex calling conventions. However, such descriptions would be repetitive and therefore more error-prone without the use of CCL symmetry. Procedure calls are highly symmetric; many actions done in the call must be "undone" in the return. CCL descriptions exploit this symmetry in both view changes and data transfers.

An action is defined to be *symmetric* if the action is performed in both an agent and the agent's symmetric agent. The caller prologue's symmetric agent is the caller epilogue, while the callee prologue's symmetric agent is the callee epilogue. Both view changes and data transfers are considered symmetric unless they are tagged with the **asymmetric** keyword.

Symmetry is used most often with view changes. When the caller prologue causes a view change by decrementing the stack pointer, the callee epilogue usually is responsible for performing the symmetric action of incrementing the stack pointer by the same amount. When this occurs, only one view change needs to be specified. The corresponding view change in the agent's symmetric agent is assumed to occur. Although symmetry may be used in data transfer statements, we have found no use for it. Initially, we had used symmetric data transfers to save and restore the values of persistent registers. In further refinements of the descriptions, we removed such symmetric data transfers because they were found to be independent of the calling convention (they define *where* in the activation record such registers should be stored, which is a calling sequence detail).

4.3.5 Summary

CCL descriptions are composed of a global section and four agent sections. These descriptions manipulate resources and sets to define the responsibilities of each of the calling convention agents. Data transfers are used to define the placement of arguments and return values. View changes are used to define how changes in the machine state changes an agent's point of view. Finally symmetry is used to reduce the amount of repetitive code by exploiting the natural symmetry of procedure calls. We conclude this section with the complete specification for the hypothetical calling convention shown in Figure 4-5.

4.4 The Formal Model

4.4.1 P-FSA Representation

We use finite state automata to model each placement in a calling convention. The use of FSA's for modeling parts of a compiler, and as an implementation tool, has a long and successful history. For example, FSA's have often been used to implement lexical analyzers [JPARG68]. More recently, Proebsting and Fraser [PF94], and Muller [Mul93] have used finite state automata to model and detect structural hazards in pipelines for instruction scheduling.

```

1.  external NVSIZE, SPILL_SIZE, LOCALS_SIZE
2.  persistent {a6, a7, a8, a9}
3.  alias sp ≡ a5
4.  caller prologue
5.    view change
6.      ∀ offset ∈ {−∞:∞}
7.        M[sp + offset] becomes M[sp + offset + ARG_SIZE]
8.    end view change
9.    data transfer (asymmetric)
10.   alias mindex ≡ sp:∞
11.   alias argregs ≡ a1:4
12.   resources {<argregs, M[mindex]>}
13.   internal ARG_SIZE ← Σ(<M[addr].size | addr ∈ <mindex> ∧
14.                        M[addr].assigned>)
15.   class regs ← <<register> | register ∈ <argregs>>
16.   class imem ← <<M[addr] | addr ∈ <sp:∞> ∧ addr mod 4 = 0>
17.   class dmem ← <<M[addr] | addr ∈ <sp:∞> ∧ addr mod 8 = 0>
18.   ∀ argument ∈ <ARG1:ARG_TOTAL>
19.     map argument → argument.type ⊥ {
20.       char:    <regs, M[mindex]>,
21.       int:     <regs, imem>,
22.       double:  <regs, dmem>,
23.     }
24.   end data transfer
25. end caller prologue
26. callee prologue
27.   view change
28.     ∀ offset ∈ {−∞:∞}
29.       M[sp + offset] becomes
30.         M[sp + offset + SPILL_SIZE + LOCALS_SIZE + NVSIZE]
31.   end view change
32. end callee prologue
33. callee epilogue
34.   data transfer (asymmetric)
35.   resources {a1:2}
36.   map RVAL1 → <<<a1>>>
37.   end data transfer
38. end callee epilogue
39. caller epilogue
40. end caller epilogue

```

Figure 4-5. A CCL description of the calling convention of Figure 4-2.

An example FSA that we use to model calling convention placement is shown in Figure 4-6. This FSA models the placement of procedure arguments for the simple calling convention. A placement FSA (P-FSA) takes a procedure's signature as input and produces locations for the procedure's arguments as output. The automaton works by moving from

state to state as the location of each value is determined. When a transition is used to move from one state to the next, information about the current parameter is read from the input, and the resulting placement is written to the output.

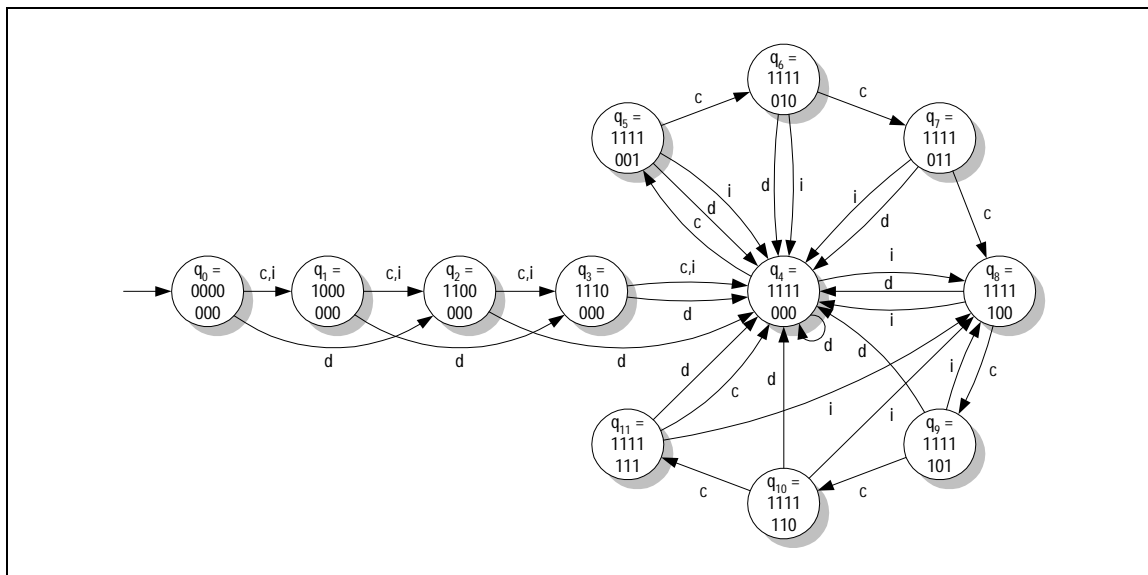


Figure 4-6. P-FSA for transmission of parameters for a simple calling convention.

The states of the machine represent that state of allocation for the machine resources. For example, the state labeled q_2 represents the fact that registers a^1 and a^2 have been allocated, but that registers a^3 , a^4 , and stack locations have not been allocated. A transition between states represents the placement of a single argument. Since arguments of different types and sizes impose different demands on the machine's resources, we may find more than one transition leaving a particular state. In our example, q_8 has three transitions even though two of them (int and double) have the same target state (q_4). This duplication is required since the output from mapping an int is different from the output from mapping a double.

Modeling the allocation of an infinite resource, such as the stack, using an FSA poses a problem, however. As mentioned above, the state indicates which resources have been allocated. For finite resources, this is easily accomplished by maintaining a bit vector. When a resource no longer may be used, the associated bit is set to indicate this. For an infinite resource this scheme cannot work if we hope to use an FSA since this would require a bit vector of infinite length. To simplify the problem, we impose a restriction on infinite resources: their allocation must be contiguous. Thus, for an infinite resource $I = \{i_1, i_2, \dots\}$, we can

store the allocation state by maintaining an index p whose value corresponds to the index of the first available resource in I . Because the allocation of I must be contiguous, p partitions the resources since a resource i_j is unavailable if $j < p$ or available if $j \geq p$. For instance, if the stack is the infinite resource, p can be considered the stack pointer.

Nevertheless, we still have a problem. Although for a particular machine, the value of p must be finite, the resulting FSA could have as many as 2^{32} stack allocation states for a 32-bit machine. However, we can significantly reduce this number by observing that the decision of where to place a parameter in memory is not based on p , but rather on alignment restrictions. For our example, we care only if the next available memory location is one-, four-, or eight-byte aligned. Consequently, we can capture the allocation state of the machine with three bits that distinguish the memory allocation states. We call these the *distinguishing* bits for infinite resource allocation.

Handling pass-by-value structures creates a complimentary problem. Since only the “alignment state” of the stack is of interest, structures that affect the state of the P-FSA differently must use different transitions. So for a convention that requires structures to be passed in 8-byte aligned memory locations, all structures of size n where $n \bmod 8 = 1$ share the same transition out of a given state because they leave the alignment, p , in the same state. Therefore, the number of transitions leaving a state is limited by the alignment restrictions of the machine.

Placement functions are described in terms of finite resources, infinite resources, and selection criteria. A set of finite resources $R = \{r_1, r_2, \dots, r_n\}$ is used to represent machine registers, while an infinite resource $I = \{i_1, i_2, \dots\}$ ¹ is used to represent the stack. The *selection criteria* $C = \{c_1, c_2, \dots, c_m\}$ correspond to characteristics about arguments (such as their type and size) that the calling convention uses to select the appropriate location for a value. We encode the signature of a procedure with a tuple $w \in (C^*, C^*)$. Each state q in the automaton is labeled according to the allocation state that it represents. The label includes a bit vector v of size n that encodes the allocation of each of the finite resources in R . Additionally, to express the state of allocation for the stack, we include d , the distinguishing bits that indicate the state of stack alignment. So, a state label is a string vd that indicates the resource allocation

1. This can easily be extended to model more than one infinite resource.

state. In our example convention, $n = 4$, and the length of the string $d(|d|)$ is 3. So, each state is labeled by a string from the language $\{0, 1\}^4\{0, 1\}^3$. The output of M is a string $s \in P$, where

$$P = R \cup \{0, 1\}^{|d|}$$

which contains the placement information.

Since the P-FSA produces output on transitions, we have a Mealy machine [Mea55].

We define a P-FSA, M , as a six-tuple¹ $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where:

- Q is the set of states with labels $\{0, 1\}^n \{0, 1\}^{|d|}$ representing the allocation state of machine resources,
- the input alphabet $\Sigma = C$, is the set of selection criteria,
- the output alphabet $\Delta = P$, is the set of placement strings,
- the transition function $\delta: Q \times \Sigma \rightarrow Q$,
- the output function $\lambda: Q \times \Sigma \rightarrow \Delta^+$, and
- q_0 is the state labeled by $0^n w$ where $|w| = |d|$, and w is the initial state of d .

We also define $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$ and $\hat{\lambda}: Q \times \Sigma^* \rightarrow \Delta^*$ which are just string versions² of δ and λ , respectively. So, for our example, we have

$$M = (Q, \{\text{char}, \text{int}, \text{double}\}, \{\text{a}^1, \text{a}^2, \text{a}^3, \text{a}^4\} \cup \{0, 1\}^3, \delta, \lambda, q_0)$$

where Q and δ are pictured in Figure 4-6 and λ is defined in Table 4-1. Note that we have modified the traditional definition of λ to allow multiple symbols to be output on a single transition. This reflects the fact that arguments can be located in more than one resource. For example, in state q_5 on an int, Table 4-1 indicates that M produces the string of four symbols 100 101 110 111 that designates four bytes that are four-byte aligned, but are not eight-byte aligned.

The signature:

```
int phred(double, double, char, int);
```

1. We use the notation of [HU79] for finite state automata and regular expressions. We use letters early in the alphabet (a, b, c) to denote single symbols. Letters late in the alphabet (w, x, y, z) will denote strings of symbols.

2. Defined by Hopcroft and Ullman [HU79].

λ	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
char	a^1	a^2	a^3	a^4	000	001	010	011	100	101	110	111
int	a^1	a^2	a^3	a^4	m_1^a	m_2^b	m_2	m_2	m_2	m_1	m_1	m_1
double	a^1a^2	a^2a^3	a^3a^4	m_3^c	m_3	m_3	m_3	m_3	m_3	m_3	m_3	m_3

Table 4-1. Definition of λ for example P-FSA.

- a. $m_1 = 000\ 001\ 010\ 011$
- b. $m_2 = 100\ 101\ 110\ 111$
- c. $m_3 = 000\ 001\ 010\ 011\ 100\ 101\ 110\ 111$

will take the P-FSA in Figure 4-6 along the path $q_0 \rightarrow q_2 \rightarrow q_4 \rightarrow q_5 \rightarrow q_4$ producing the string $(a^1\ a^2)\ (a^3\ a^4)\ (000)\ (100\ 101\ 110\ 111)$ along the way. The parentheses in the output string are required to determine where the placement of one argument ends and the next argument's placement begins. From the string, we can derive the placement of the phred's arguments. The first double is placed in registers a^1 and a^2 , the second in registers a^3 and a^4 , the char at the stack location with offset zero and the int at the stack location with offset four.

4.4.2 Automatic P-FSA Construction

In this section, we present an algorithm for automatically constructing P-FSAs. For the moment, we assume the existence of a function $f: \Sigma^* \rightarrow \Delta^*$. f computes the same value as M . Since f and M are equivalent, why construct M at all? The answer is that f may have undesirable properties. For instance, M may be used in a context, such as a compiler, where performance is an issue. If f is implemented as an interpreter, the time it takes to compute a placement may not satisfy the performance constraints. Additionally, by using a P-FSA, there are several properties (such as an upper bound on M 's execution time) we can prove about the P-FSA that we cannot prove about f .

We construct the P-FSA by performing a depth-first-traversal of the states in Q to determine the set of reachable states from q_0 . At each state q , the states that are reachable from q in one step are determined by using each element of $\{wc \mid c \in C\}$ as input to f . Each newly reachable state q' is added to Q and is subsequently visited by BUILD-P-FSA (see Figure 4-7). Finally, the appropriate additions to δ and λ are made for q' . BUILD-P-FSA also uses an auxiliary function $\text{STATE-LABEL}: P \rightarrow Q$. STATE-LABEL takes an output string from M and computes the label for the state that M was in when the input was exhausted.

4.4.2.1 Construction Algorithms

We define the algorithm Build-P-FSA in Figure 4-7. The algorithm starts with the initial state q_0 as the only element of Q . Since there are no transitions yet, λ and δ have no rules. A call to Build-P-FSA takes three parameters, q , w , and x . q represents the state for Build-P-FSA to visit, while w represents the input string such that (q_0, w) yields (q, ε) , and x is output string upon reaching q . From this definition, the initial call to Build-P-FSA must be Build-P-FSA($q_0, \varepsilon, \varepsilon$).

```

function BUILD-P-FSA( $q, w, x$ )
  //  $q \in Q, w \in \Sigma^*, x \in \Delta^* \mid \lambda(w) = x$ 
  for each criterion  $c \in C$  do
     $y \leftarrow f(wc)$ ; // compute placement of signature  $wc$ 
     $q' \leftarrow \text{STATE-LABEL}(y)$ ; // compute state label from placement
    if  $q' \notin Q$  then
       $Q \leftarrow Q \cup \{q'\}$ ;
      BUILD-P-FSA( $q', wc, y$ );
    end if
     $a \leftarrow b \mid xb = y$ ; // set  $a$  as the suffix of  $y$  not in  $x$ 
    add  $\lambda(q, c) = q'$ ;
    add  $\delta(q, c) = a$ ;
  end for
end function

```

Figure 4-7. Algorithm to build a P-FSA.

In the algorithm for STATE-LABEL we start with state q_0 . As STATE-LABEL reads each symbol from the string, it encounters either the name of a finite resource, or a symbol representing the distinguishing bits of p . In the finite case, the bit corresponding to the resource is set in the finite resource vector. In the infinite case, the distinguishing bits of the state are set to the input symbol that was read. At the end of the input, all finite resources that have been read have their bits set to indicate they are unavailable, and the distinguishing bits indicate the last set of distinguishing bits read. To complete the computation, we need to move the infinite resource index to the next available resource (it currently points to the last unavailable one)¹. The result of this computation is precisely the label for the final state of M for output w since it indicates which resources are available for allocation. The complete algorithm is shown in Figure 4-8.

1. An ordered list of values for p 's distinguishing bits is known so that we can perform this calculation, although this is usually just an increment.

function STATE-LABEL(w)	// $w \in \Delta^*$
$z \leftarrow 0^n$;	// z is the finite resource vector
while $w \neq \varepsilon$ do	
// extract the first symbol from w	
define a and x such that $ax = w$;	
$w \leftarrow x$;	// set w to the rest of w
if $a \in R$ then	// for finite resources:
// mark it as used	
set a 's corresponding bit in z ;	
else	// for infinite resources:
$d \leftarrow a$;	// keep the last one encountered
end if	
end while	
$d \leftarrow d + 1$;	// set d to the next resource
return zd ;	// return state label made of z and d
end function	

Figure 4-8. Definition of STATE-LABEL.

Our construction is now complete, except the definition of the function f . We supply f 's definition using an interpreter. The interpreter takes as input a CCL specification, information about a procedure's signature and some additional information about the target machine, and produces the necessary mapping information to properly call the given procedure. Thus, this interpreter can be used to implement f in our algorithm above. In Section 4.5.1, we present the interpreter's use in an implementation.

4.4.3 Completeness and Consistency in P-FSA's

Applications, such as compilers and debuggers, which generate, or process procedures at the machine-language level require knowledge of the calling convention. Until now, the portion of such an application's implementation that concerned itself with the procedure call interface was constructed in an *ad-hoc* manner. The resulting code is complicated with details, difficult to maintain, and often incorrect. In our experience, we have encountered many recurring difficulties in the calling convention portion of a retargetable compiler. There are three sources for these problems: the convention specification, the convention implementation, and the implementation process. We address each of these in the following paragraphs.

Many problems arise from the method of convention specification. Often, no specification exists at all. Instead the native compiler uses a convention that must be extracted by reverse engineering. In the cases where a specification exists, it typically takes the form of written prose, or a few general rules (*e.g.*, our example description in Figure 4-2). Such methods of specification have obvious deficiencies. Furthermore, even if we have an accurate method for

specifying a convention, it still may be possible to describe conventions that are internally inconsistent, or incomplete. For example, the convention may require that more than one procedure argument be placed in a particular resource. Another possibility is that the specification may omit rules for a particular data type, or combination of data types.

Those problems that do not stem from the specification result from incorrect implementation of the convention. Many of the same problems in the specification process also plague the implementation. Many conventions have numerous rules, and exceptions that must be reflected in the implementation. Another difficulty is that the implementation may require the use of the convention in several different locations. Maintaining a correspondence between the various implementations can itself be a great source of errors. Finally, this problem is exacerbated by the fact that the implementation frequently undergoes incremental development. Rather than taking on the chore of implementing the entire convention at once, a single aspect of the convention, such as providing support for a single data type, is tackled. After successfully implementing this subset, the next increment is tackled. During this process, some aspect of the first stage may break due to the interactions between the two pieces.

The result of these observations is that there are several properties that we would like to ensure about a specification and implementation. The above discussion motivates the following categories of questions:

- Completeness:
 - *Does the specified convention handle any number of arguments?*
 - *Does the convention handle any combination of argument types?*
- Consistency:
 - *Does the convention map more than one argument to a single machine resource?*
 - *Do the caller and callee's implementations agree on the convention?*

Many questions like these can be answered using P-FSA's. The following sections show how we can prove certain properties about CCL specifications that ensure desirable responses to the above questions.

4.4.3.1 Completeness

The completeness properties address how well the convention covers the possible input cases. A convention must handle any procedure signature. If we could guarantee that the convention was complete, or covered the input set, then we could answer the completeness questions

posed in the previous section. We can determine if a convention is complete by looking at the resulting P-FSA. For example, will the convention work for any combination of argument types? The answer lies in the P-FSA transitions. For the convention to be complete, each state $q \in Q$ must have $\delta(q, c)$ defined for all $c \in C$.

Using P-FSA's, we can guarantee that no incomplete convention will go undetected. For an incomplete convention K to not be detected, it would first have to be constructed using our algorithm. Assume such a P-FSA M exists for K . Then there must be some state q_k that is reachable from q_0 but does not have $\delta(q_k, a)$ defined for some $a \in C$. Let W_k denote the set of all strings x such that $\hat{\delta}(q_0, x) = q_k$. That is, W_k is the set of strings that take M from state q_0 to q_k . Thus, for all strings x such that $x \in W_k$, xa represents a signature that K does not cover. However, during construction, BUILD-P-FSA visited state q_k with some string w such that $\hat{\delta}(q_0, w) = q_k$. Thus, w must be in W_k and must not be covered by K . Since BUILD-P-FSA calls $f(wc)$ for all $c \in C$, f will be called using $f(wa)$. Since wa is not covered by K , $f(wa)$ will be undefined. At this point the construction process will signal that K is incomplete.

4.4.3.2 Consistency

The consistency properties address whether the convention is internally and externally consistent. A convention is internally consistent if there is no machine resource that can be assigned to more than one argument. A convention is externally consistent if the caller and callee agree on the locations of transmitted values. In our model, we *detect* internal inconsistency, and *prevent* external inconsistency.

To detect internal inconsistencies, we again turn to the P-FSA. If the convention only used finite resources, detecting a cycle in the P-FSA would be sufficient to detect the error. However, when infinite resources are introduced, so are cycles. We cannot have an internal inconsistency for an infinite resource since p is defined to be monotonically increasing. We detect finite resource inconsistencies in the following manner. An inconsistency can occur when there is a transition from some state q_j to q_k where bit i in the finite bit vector is 1 in q_j but 0 in q_k . At this point, M has lost the information that resource r_i was already allocated.

We can detect this change by comparing all pairs of bit vectors v_1 , v_2 such that v_1 labels q_j , v_2 labels q_k and $\delta(q_j, c) = q_k$ for some $c \in C$. To do the comparison, we compute

$$v_3 = (v_1 \oplus v_2) \wedge v_1$$

$v_1 \oplus v_2$ selects all bits that differ between v_1 and v_2 . We logically AND this with v_1 to determine if any set bits change value. Thus, if v_3 has any bit set, we have an inconsistency.

Our convention specification language prevents external inconsistencies in the calling convention. A convention specification only defines the argument transmission locations once. Although both the caller and the callee must make use of this information, the specification does not duplicate the information. Since we only have a single definition of argument locations, we only construct a single P-FSA to model the placement mapping. This single P-FSA is used in both the caller and callee. Thus, we prevent external inconsistencies by requiring the caller and callee use the same implementation for the placement mapping.

4.5 Use in a Compiler

In this section, we present how the information from our CCL descriptions can be used to generate calling sequences for an optimizing compiler.

4.5.1 The Interpreter

We have implemented an interpreter for the CCL specification language. The interpreter's source is approximately 2500 lines of Icon code [GG90]. The interpreter takes as input the CCL description of a procedure calling convention, a procedure's signature, and some additional information about the target architecture, and produces locations of the values to be transmitted, in terms of both the callee and the caller's frame of reference.

We have developed CCL specifications for the following machines: MIPS R3000 [KH92], SPARC [Sun87], DEC VAX-11 [Dig78], Motorola M68020 [Mot85], and Motorola M88100 [Mot88]. Each of these CCL specifications is approximately one page in length. Using the specification for the MIPS, and the CCL interpreter, we constructed a P-FSA that implements the MIPS calling convention. The MIPS P-FSA uses only 70 out of a possible 512 states (the state label has nine bits), but requires up to 25 transitions for each state to implement the selection criteria for the C programming language. Since the MIPS convention has more machine resource classes and alignment requirements than any of the other

machines, it represents the most complicated convention we have. For machines that pass procedure arguments on the stack with no alignment restrictions, such as the VAX-11, the FSA's contain only a few states.

For comparison purposes, we have examined the calling convention specific code for a retargetable compiler. The MIPS implementation requires 781 lines of C code, while the SPARC implementation has 618 lines. This code is one of the most complex sections of the machine-dependent code. This code is replaced by the P-FSA tables and a simple automaton interpreter.

4.5.2 Realizing the Calling Sequence

In our compiler, the code for the procedure bodies is generated without knowledge of the calling convention. For a callee, the optimizer treats formal parameters as local variables. It assigns each parameter either a register or a memory location, based on the parameter's predicted reference frequency. Thus, although an established convention for where values cross the procedure call interface exists, the code generated by our compiler for a procedure's body may not conform to the convention.

To correct this problem, instructions are placed before and after the callee's body, and before and after the call site in the caller. We call these instructions the caller/callee prologue/epilogue sequences. It is these sequences of instructions that are collectively called the calling sequence. The sequences introduce four new interfaces shown in Figure 4-9. In each sequence, the instructions transform a convention interface to a code body interface or *vice versa*. Since these sequences of instructions are used to "attach" the procedure bodies to the convention interfaces, they correspond to the agents, shown in Figure 4-3, of our high-level model.

An agent's responsibilities fall into each of three categories: allocation or deallocation of storage space, movement of values from their locations in the first interface to locations in the second interface, and the construction/restoration of procedure execution environments. Hence, to generate an agent's actions, we must have information about where the calling convention expects values, what space to allocate or free, and the procedure's environment structure. We can automatically generate the first two.

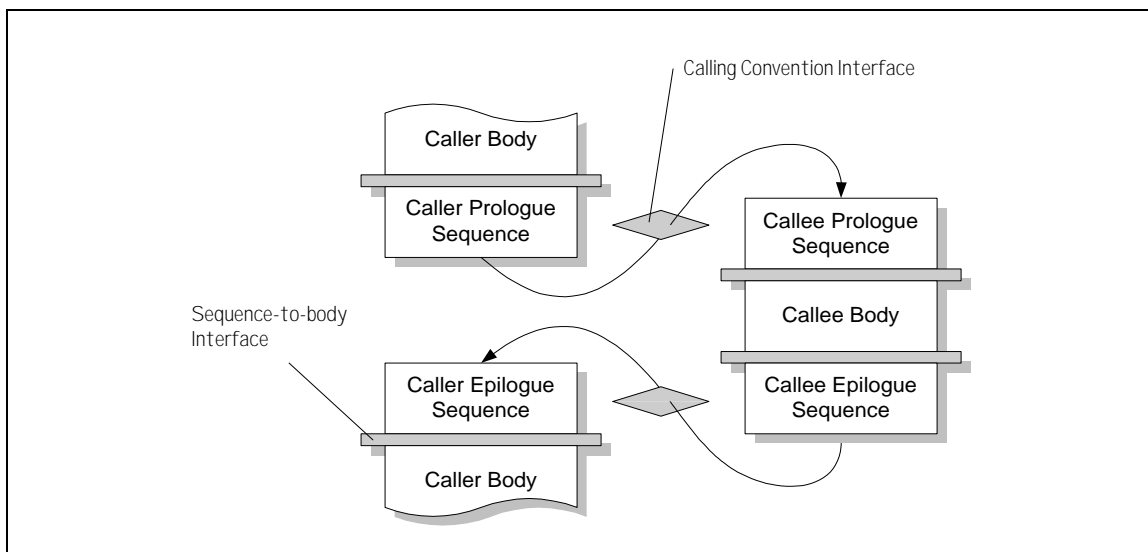


Figure 4-9. Calling sequence locations.

To illustrate our technique, we show how to generate the instruction sequence for one agent. The instruction sequences that correspond to the other three agents are generated exactly the same way. For our example, we focus on the prologue callee agent for the procedure warp introduced in Section 4.2.1. Again, warp's signature:

```
int warp(char p1, int p2, int p3, double p4);
```

Recall that for our hypothetical machine, warp's arguments are placed by the caller in locations a^1 , a^2 , a^3 , $M[\text{sp}:\text{sp}+7]$. Assume that in generating warp's body, the optimizer uses two persistent registers, allocates 12 bytes of memory for local variables (including warp's arguments) and uses eight bytes of spill space. One possible frame layout is shown in Figure 4-10. Figure 4-10(a) shows the generic layout for any procedure, while Figure 4-10(b) shows warp's layout using this scheme. The relative locations of the temporary spill space, local variable space and persistent register save space are determined by the optimizer. The optimizer provides the locations where the callee body expects values. These are listed in the second column of Table 4-2. These locations represent an agreement between the callee body and the callee prologue agent.

The optimizer calls the P-FSA interpreter with warp's signature and values of the external variables:

```
[SPILL_SIZE=8, LOCALS_SIZE=12, NVSIZE=8,
 (ARG1, type:char, size:1), (ARG2, type:int, size:4), (ARG3, type:int, size:4),
 (ARG4, type:double, size:8)]
```

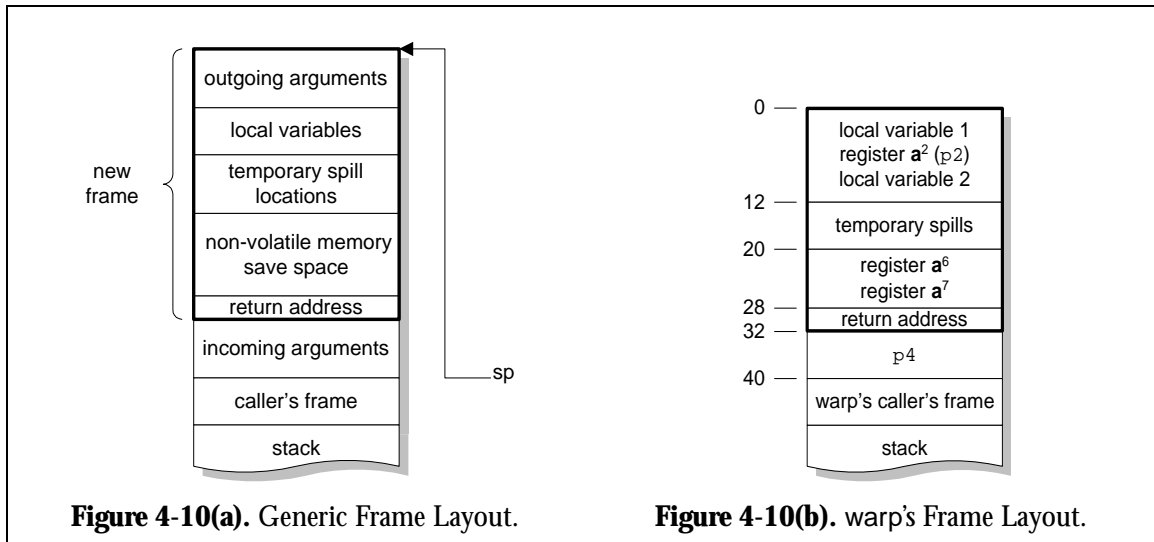


Figure 4-10(a). Generic Frame Layout.

Figure 4-10(b). warp's Frame Layout.

Figure 4-10. A possible procedure activation frame structure.

The P-FSA returns view changes, a list of argument locations that correspond to the calling convention, and a list of persistent registers:

$$\begin{aligned}
 &[(\forall \textit{offset} \in \{-\infty:\infty\}, \textit{M}[\textit{sp} + \textit{offset}] : \textit{M}[\textit{sp} + \textit{offset} + 32]), \\
 &[(\textit{ARG}^1, \textit{a}^1), (\textit{ARG}^2, \textit{a}^2), (\textit{ARG}^3, \textit{a}^3), (\textit{ARG}^4, \textit{M}[\textit{sp}+32:\textit{sp}+39])], \\
 &[\textit{persistent}: \textit{a}^6, \textit{a}^7, \textit{a}^8, \textit{a}^9]]
 \end{aligned}$$

In this example, the view change occurred before the list of locations. Therefore, the locations reflect this fact.

View change information corresponds to the allocation or deallocation of storage space. This view change indicates that any memory location's address that contains a valid value for offset, shifts down by 32 bytes. Since offset can take on any positive or negative value ($-\infty:\infty$), this corresponds to all addresses relative to the stack pointer. Thus, a decrement of the stack pointer by 32 bytes is needed. This allocation of stack space will appear as a view change since it changes the names of all locations referenced by the stack pointer. A table is consulted for each view change in the CCL description. The table maps all view changes to valid machine instructions.

After the view change has been performed, the necessary moves must be made to transform the agreement between the caller prologue agent and callee prologue agent to the agreement between the callee prologue agent and the callee body. Table 4-2 summarizes the location information. Column one shows the locations returned by the P-FSA. Column two shows the locations that the optimizer supplies. Column three, which can be trivially derived

from columns one and two, indicates the necessary actions. Each of these moves is a register/memory to register/memory move. A table of available move instructions is consulted to determine the necessary instructions to be inserted into the callee prologue's sequence.

After the agent's actions are determined, the list of sources and destinations must be examined to determine if there are any dependencies. If a source is also a destination, the move containing the source must be performed before the move containing the destination, otherwise the source value will be lost. It is not uncommon for a circularity to exist. For example, if $a^1 \rightarrow a^2$ and $a^2 \rightarrow a^1$, we must introduce a third location to break the circularity: $a^1 \rightarrow \text{temp}$, $a^2 \rightarrow a^1$, $\text{temp} \rightarrow a^2$. Either an available register or a memory location must be used to temporarily hold one of the values. In our compiler, we usually have a register available.

	Convention	Callee Prologue Agent/Callee Agreement	Callee Prologue Agent Actions
Arguments	$p1:a^1$	$p1:a^3$	$a^1 \rightarrow a^3$
	$p2:a^2$	$p2:M[\text{sp}+4:\text{sp}+7]$	$a^2 \rightarrow M[\text{sp}+4:\text{sp}+7]$
	$p3:a^3$	$p3:a^4$	$a^3 \rightarrow a^4$
	$p4:M[\text{sp}+32:\text{sp}+39]$	$p4:a^1, a^2$	$M[\text{sp}+32:\text{sp}+39] \rightarrow a^1, a^2$
Persistent	a^6	$M[\text{sp}+20:\text{sp}+23]$	$a^6 \rightarrow M[\text{sp}+20:\text{sp}+23]$
	a^7	$M[\text{sp}+24:\text{sp}+27]$	$a^7 \rightarrow M[\text{sp}+24:\text{sp}+27]$
	a^8	a^8	—
	a^9	a^9	—

Table 4-2. Determining agent actions from placement information.

At this point, the callee prologue instruction sequence is complete. So far, we have not addressed instruction sequence efficiency. Because of the frequency of procedure calls, generating efficient instruction sequences is an important feature of optimizing compilers. In our compiler, the resulting instruction sequences are processed by the optimizer. Thus, although the instruction sequences that are initially generated by this process are naive, they benefit from thorough optimization just as other code does. The resulting code is as good, if not better, than the code generated by our handwritten version of our compiler. Often, the code

improves because the additional peephole optimization phase that is performed after the calling sequence instructions are generated can remove unnecessary register-to-register moves.

Clearly, P-FSA's are very useful for generating code in compilers. We have shown how we can use CCL descriptions to build P-FSA's that can subsequently be used in the implementation of a compiler. Code based on such formalisms has many advantages. However, the usefulness of P-FSA's are not limited to code generation. In the next section, we illustrate how P-FSA's are used to build target-sensitive test suites for compilers. Using CCL descriptions for such a variety of applications is only possible because CCL specifications exhibit such a degree of application-independence.

4.6 Construction of Diagnostic Programs

Building compilers that generate correct code is difficult. To achieve this goal, compiler writers rely on automated compiler building tools and thorough testing. Automated tools, such as parser generators, take a specification of a task and generate implementations that are more robust than hand-coded implementations. Conversely, testing tries to make hand-coded implementations more robust by detecting errors. In this section, we discuss how CCL descriptions can be used to make compilers more robust without requiring that the compiler's implementation use CCL [BD96b].

4.6.1 Test Vector Selection

To test a compiler's implementation of a calling convention, we must select a set of programs to compile. To exercise the calling convention, each test program must contain a caller and a callee procedure. For the purpose of testing the proper transmission of program values between procedures, the signature of the callee uniquely identifies a test case. Thus, two different programs, whose callees' signatures match, perform the same test. Therefore, the problem of generating test cases reduces to the problem of selecting signatures to test.

Selecting which procedure signatures to test is a difficult problem. Obviously, one cannot test all signatures since the set of signatures, $S = \{(C^*, C^*)\}$, is infinite. However, since we can model the function that computes the placement of arguments as an FSA, there must be a finite number of states in an implementation to be tested. This is the case for any implementation, including those that do not explicitly use FSA's to model the placement function.

The problem of confirming that an implementation properly places procedure arguments is equivalent to experimentally determining if the implementation behaves as described by the P-FSA state table. This problem is known as the *checking experiment problem* from finite-automata theory [Hen64, Koh78]. There are numerous approaches to this problem, most of which are based on transition testing. Transition testing forces the implementation to undergo all the transitions that are specified in the specification FSA.

An obvious first approach to generating test vectors using the P-FSA specification is to generate all vectors whose paths through the FSA are acyclic and those whose path ends in a cycle¹. This solution insures that each state q is visited, and each transition $\delta(q, a)$ is traversed. For an FSA with few states, and a small input alphabet, this may be acceptable. However, the number of such paths for an FSA is $O(|\Sigma|^{|Q|})$. To illustrate the characteristics of P-FSA's, Table 4-3 contains profiles for five P-FSA's that we have built from CCL descriptions. For complex conventions, like the MIPS and SPARC, the number of transitions, and more important, the number of states can be large. For the MIPS, this results in an upper bound of $25^{12} = 2.3 \times 10^{22}$ test vectors. In practice, the number of test vectors is closer to 10^8 vectors. However, this is still too many to run feasibly.

Machine	Allocation Vector Bits	Memory Partition Bits	$ Q $	$ \delta $	$ \Sigma $	Longest Acyclic Path
DEC VAX	0	0	1	3	3	0
M68020 (Sun)	0	2	4	24	6	3
SPARC (Sun)	6	3	9	90	10	8
M88100 (Motorola)	8	3	72	720	10	15
MIPS R3000 (DEC)	6	3	70	772	25	11

Table 4-3. P-FSA profiles for several calling conventions.

Another, simpler approach is to guarantee that each transition is exercised at least once. Since there are no more than $|Q| \cdot |\Sigma|$ transitions, the number of test vectors that this generates is not unreasonable. However, this method results in poor coverage that does not

1. We define a *path that ends in a cycle* to be a cyclic path wa where the path w is acyclic.

inspire confidence in the test suite. For example, for the P-FSA in Figure 4-6, the three signatures:

```
void f(double, double);
void f(int, int, int, int);
void f(int, double);
```

cover all `int` and `double` transitions leaving states q_{0-2} . This leaves the signature:

```
void f(double, int);
```

untested. Clearly such a test should be included in the suite. To further illustrate the problem, consider the FSA specification shown in Figure 4-11(a). An erroneous implementation, shown in Figure 4-11(b), contains an extra state q_1' that is reached on initial input `b`. The two strings, `aaa` and `bbb` completely cover the specification FSA transitions. Unfortunately, these test vectors will not detect that the implementation has an additional (fault) state. Thus, it is not sufficient to include only test vectors that cover the transition set.

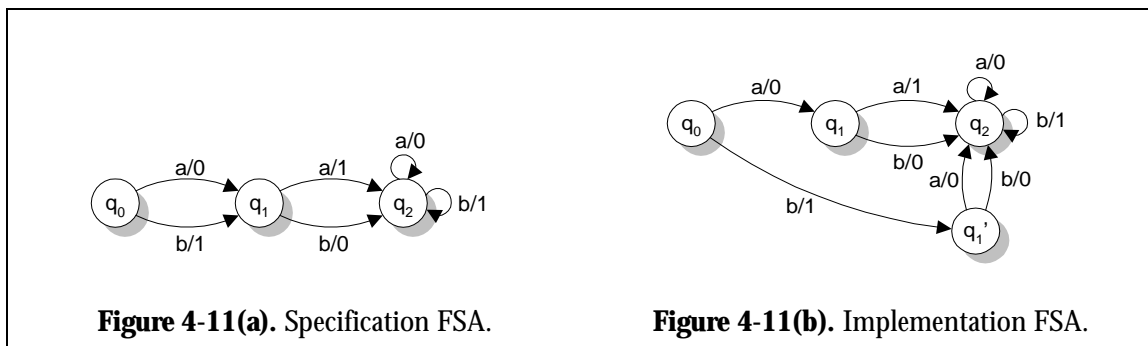


Figure 4-11. Example FSA where a fault will not be detected.

An alternative, which falls between the simple transition approach and the acyclic path approach, we call the *transition-pairing* approach. In transition pairing, we examine each state in the specification FSA. As shown in Figure 4-12, a state has entering and exiting transitions. For each state, we include a test vector that covers each *pair* of entering and exiting transitions. This eliminates the faulty state detection problem illustrated in Figure 4-11. To illustrate how, consider the test vectors this process generates: While examining state q_1 , transition-pairing will add the substrings `aa`, `ab`, `ba`, and `bb` to the set of substrings used to generate test vectors. Since the context that these substrings are be used is q_0 , they contribute prefixes to the test vector set. Upon exercising q_1 using the prefix `ba`, the implementation FSA

will generate incorrect output: 10 instead of 11. This difference can be identified, and the faulty state detected.

In addition to such fault detection, transition-pairing provides tests that have a similar characteristic to the acyclic method: transitions are tested in “all” the contexts that they can be applied. Although there are many combinations that are not tested, they are similar to ones included in the set. For example, in the simple FSA pictured in Figure 4-6, we could have a set of test vectors that includes the vector double double double to exercise the state q_4 with the transition pair $((q_2, \text{double}), (q_4, \text{double}))$. Such a set would not need to include int int double double to cover the same transition pair.

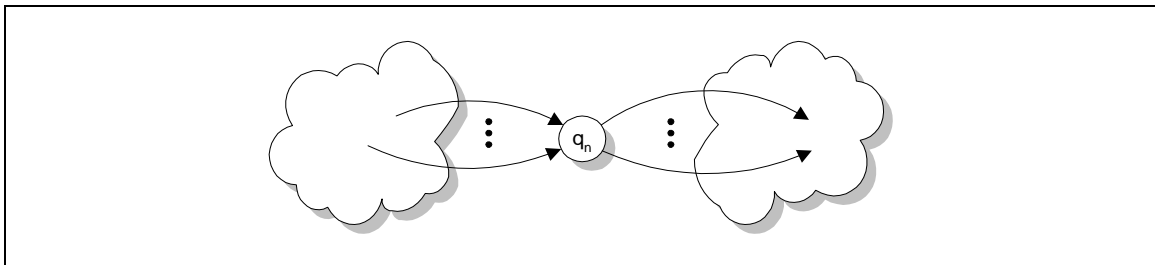


Figure 4-12. Entering and exiting transitions for a state.

This method of test vector generation provides a complete coverage of transitions in the specification FSA. Further, the tests reflect the context sensitivity that transitions have. This allows for some erroneous state and transition detection, while significantly reducing the number of test vectors. The test vector sizes are significantly smaller than the acyclic method, while still providing a significant confidence level.

Machine	Transition Paths	Transition-Pair Paths	Acyclic Paths
DEC VAX	3	12	3
M68020 (Sun)	24	324	96
SPARC (Sun)	224	7,434	$> 10^8$
M88100 (Motorola)	720	22,412	$> 10^8$
MIPS R3000 (DEC)	772	5,655	8×10^8

Table 4-4. Sizes of test suites for various selection methods.

An algorithm for generating transition-pair paths is shown in Figure 4-13. The algorithm performs a depth-first search of the FSA state graph. Each time a transition (q, a) is

encountered, it is marked. This mark indicates that all paths that go beyond (q, a) have been visited. When the algorithm reaches a state q_n on transition (q_m, a) , each transition (q_m, b) where $b \in \Sigma$ is visited whether or not it is marked. This causes all pairs of transitions $((q_m, a), (q_m, b))$ to be included. These pairs represent all combinations of one entering transition with all exiting transitions. Because the algorithm is depth-first, each entering transition is guaranteed to be visited. Thus, all combinations of entering and exiting transitions are included.

```

Input. A finite-state machine  $M$ .

Output. The set of transition-pair paths in  $M$  that take  $M$  from  $q_0$  to  $q_n$  with at most one cycle. The set
traverses all pairs of transitions  $((q_r, a), (q_s, b))$  such that  $\delta(q_r, a) = q_s$ .

Initial call. TRANSITION-PAIRS( $q_0, \epsilon, \emptyset, 0$ );

Algorithm.
function TRANSITION-PAIRS( $q, w, V, \text{cycle}$ )
  paths  $\leftarrow \emptyset$ ;
  for each  $a$  where  $a \in \Sigma \wedge \delta(q, a)$  is defined do
    if  $\text{cycle} \neq 1 \wedge (q, a) \notin T$  then
      if  $q \notin V$  then
         $T \leftarrow T \cup \{(q, a)\}$ ;
         $\text{cycle} \leftarrow 0$ ;
      else
         $\text{cycle} \leftarrow 1$ ;
      end if
       $P \leftarrow \text{TRANSITION-PAIRS}(\delta(q, a), wa, V \cup \{q\}, \text{cycle})$ ;
      paths  $\leftarrow \text{paths} \cup P$ ;
    end if
  paths  $\leftarrow \text{paths} \cup \{wa\}$ ;
end for
return paths;
end function

```

Figure 4-13. Test vector generation algorithm.

Work related to the automatic generation of test suites has received much attention recently in the area of conformance testing of network protocols [SL89]. The purpose of these suites is to determine if the implementation of a communication protocol adheres to the protocol's specification. Often, the protocol specification is provided as a finite-state machine. This has resulted in many methods of test selection including the Transition tour, Partial W-method [FvBK+91], Distinguishing Sequence Method [Koh78], and Unique-Input-Output method [ADLU91]. These methods are derivatives of the checking experiment problem

where an implementation is checked against a specification FSM [YL95]. Such techniques have also been used in the automatic verification of digital circuits [Hen64, HYHD95]

What distinguishes these methods from ours are the underlying assumptions concerning the characteristics of the implementation FSA's. Unlike theirs, our FSA's can have a large number of states and transitions. This significantly changes the nature of the solution to the problem. Furthermore, much of the problem that network conformance researchers are faced with is identifying which state the implementation FSA is in. A significant portion of their work focuses on generating test vectors that discover the state of the machine. Fortunately, we can always put our implementation state machine in the start state. Also, in their work, a bound on the number of states in the implementation FSA's is assumed. Because we have no practical bound on the number of states in the implementation, their work is not applicable.

4.6.2 Test Case Generation

After selecting the appropriate test vectors, or procedure signatures, the corresponding test cases must be realized. In our approach, we generate a separate test program for each test vector so that we can easily match any reported errors to the specific test vector.

A procedure call is broken into two pieces: the procedure call within the caller (the call-site) and the body of the callee. Because they are implemented differently, these two pieces of code are typically generated in separate locations in a compiler. This natural separation is reflected in the way that we construct our test cases. Each test case is comprised of two files, one contains the caller, the other contains the callee. The two files are compiled and linked together. The programs are self-checking, so that if a procedure call fails, this event is reported by the test itself.

Figure 4-14 shows the compiler conformance test process. One file is compiled by the compiler-under-test (CUT), while the other is compiled by the reference compiler. The reference compiler operationally defines the procedure calling convention (its implementation is defined to be correct). The resulting objects files are linked together and run. Results of the test are checked by the conformance verifier and given to the test conductor. The test conductor tallies the results of all tests for a test suite and generates a conformance report. Although this process uses two compilers, the same process may still be used if a reference compiler is

not available. However, this will weaken the conformance verifier's ability to automatically diagnose errors as discussed in the next section.

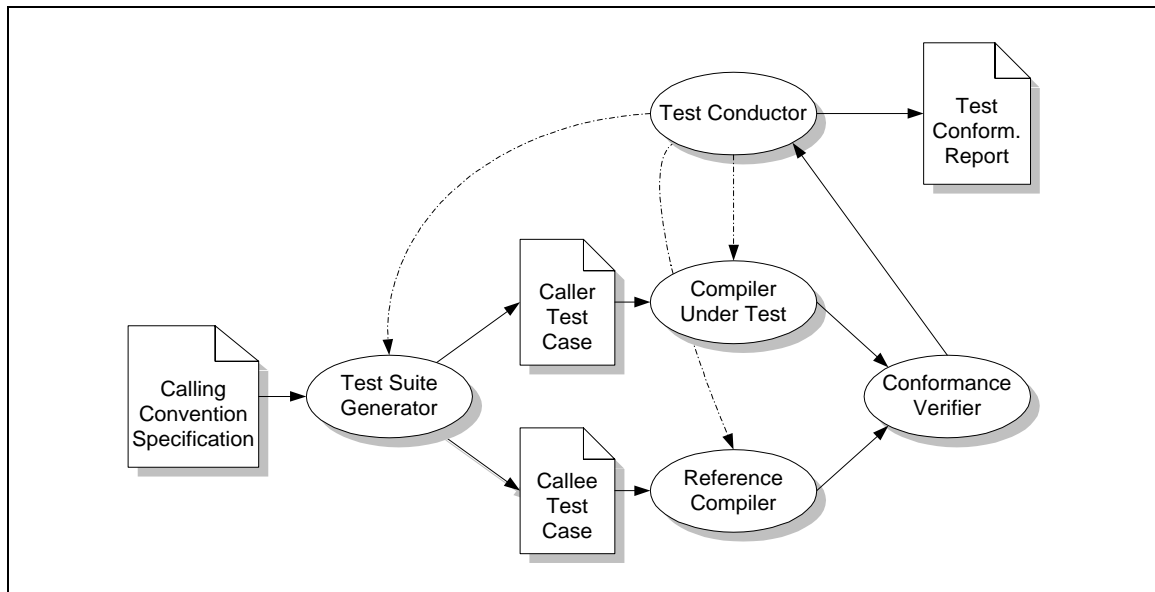


Figure 4-14. The compiler conformance test process.

In each test case, the caller loads each argument with randomly selected bytes. However, the values of these bytes have an important property: each contiguous set of two bytes is unique. Thus, for a string B of m bytes, for all indexes $0 < i \leq m$, there exists no index $0 < j \leq m$ and $j \neq i$ such that $B[j+k] = B[i+k]$ for all $0 \leq k < 2$. We can easily guarantee this property for all strings B whose length is no more than 65536 (2^{16}) bytes. Since the likelihood of using an argument list of size greater than 64 Kbytes is small, this is sufficient to guarantee that any two bytes passed between procedures are unique. This makes it easier to identify if an argument has been shifted or misplaced. The callee receives the values, and checks them against the expected values. If the values do not match, an error condition is signalled.

As one might expect, the generation of good test cases from selected signatures is language dependent. One convention used in the C programming language is *varargs*. *varargs* is a standard for writing procedures that accept variable length argument lists. The proper implementation of *varargs* in a C compiler is difficult. For each test case that we generate we also generate a *varargs* version to verify that this standard convention is implemented correctly.

4.6.3 Automatic Diagnosis of Errors

Generation of good tests is only a part of the testing process. If a test fails, the problem must be diagnosed and a solution developed. In this section, we discuss how the second step, diagnosis, can be partially automated.

As discussed above, the conformance verifier links a caller and callee together and runs the resulting program. When both a reference compiler and CUT are used, this results in four distinct caller-callee pairs. The result of running all four programs is called an *outcome*.

Figure 4-15 shows an outcome graphically. Procedures generated by the reference compiler are filled, while CUT generated components are unfilled. The result of a single test is indicated by an arrow connecting a pair of components. When the result is that a test passed, a solid line is shown, while a dotted line is used for test failure.

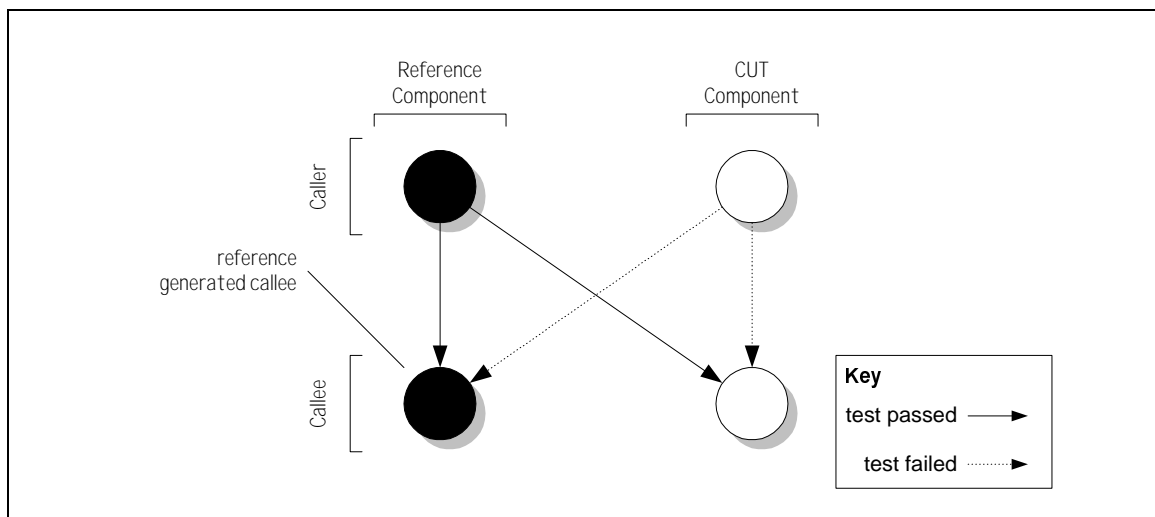


Figure 4-15. An example outcome.

The result of a single test, taken in isolation, provides limited information: whether a fault has been detected or not. However, we can glean more information by considering the composite result that an outcome provides. By using multiple versions of object files generated by different compilers, we can exploit the interface of the procedure call. Each test has an object file in common with two other tests. When a test fails, the results of the two other tests can help isolate the fault. For example, in the outcome shown in Figure 4-15, the CUT/reference test (the test comprised of the CUT caller and reference callee) has failed. To isolate if the caller or callee contains the fault, the reference/reference test result is considered. This test replaces the CUT caller with the reference caller, keeping the callee in common between the

two tests. Since the test passed, we have reason to believe that the CUT caller contains the fault since the fault disappeared when the CUT caller was removed. Our suspicion is confirmed when we consider the CUT/CUT test. Since this test fails, the fault remains when the reference callee was removed. Thus, the fault must be in the CUT caller. We would come to the same conclusion had we started with the CUT/CUT fault and considered the CUT/reference and reference/CUT test results.

This method of isolating errors by swapping different components makes it possible to automatically diagnose common errors. Since each outcome is comprised of four results that may indicate a pass or fail, there are 16 outcome configurations. Since this number is small, each outcome can be hand-analyzed once and the results tabulated. Table 4-5 summarizes such an analysis. Several diagnoses deserve mention. First, although the reference compiler is considered the authority, there are many cases where the reference can be determined to be faulty. This occurs in six of the outcomes. Second, three of the outcome configurations are not possible. These are the outcomes where only a single test failed. This indicates a conflict in conventions. This cannot occur with a single test failure since we assume each component uses a single convention¹. Finally, for two of the cases, we not only can isolate the location of the fault, but we can identify the nature of the error. This occurs in outcomes D and M where two conflicting conventions have been discovered.

The combination of test vector selection and automatic diagnosis proves to be a powerful debugging tool. As tests are generated, run, and analyzed, patterns of errors tend to emerge. We have found that the patterns themselves suggest the nature of the problem. For example, finding that an error occurred for every signature that included a struct of size greater than seven bytes might suggest an alignment problem. More complicated patterns can exist, and, with knowledge of the calling convention can significantly help the developer correct faults.

1. Appel observes that such outcomes actually are possible [App96]. In his counter example, the CUT caller implements a different convention than the reference compiler, but the CUT callee implements *both* conventions. In this scenario, the fault is detected in the CUT/reference test, but not in either the CUT/CUT or the reference/reference tests. Although such a case is possible, the chances of a callee implementing two different conventions that do not conflict (*i.e.*, use the same register for two different purposes) are remote. The benefits, in terms of diagnostic ability, of considering such a case as invalid, far outweigh any accuracy gained by labeling it a valid outcome. Finally, if such a case were to occur, it would still be detected; it just could not be automatically diagnosed.

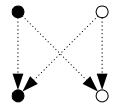
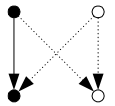
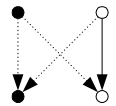
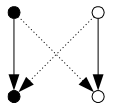
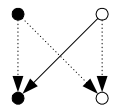
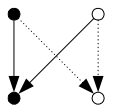
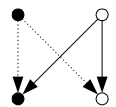
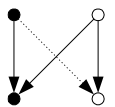
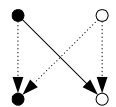
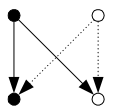
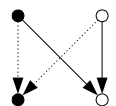
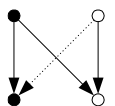
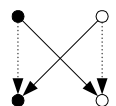
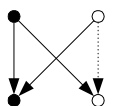
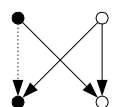
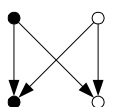
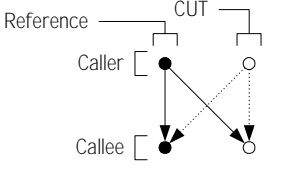
Outcome	Diagnosis	Outcome	Diagnosis
	Outcome A: Faults in at least three components.		Outcome B: Faults in both components of the CUT.
	Outcome C: Faults in both components of reference compiler.		Outcome D: CUT implements wrong convention (does not externally conform with the reference).
	Outcome E: Fault in the reference compiler's caller. Fault in the CUT's callee.		Outcome F: Fault in the CUT's callee.
	Outcome G: Fault in the reference compiler's caller.		Outcome H: Not a possible outcome.
	Outcome I: Fault in reference compiler's callee. Fault in CUT's caller.		Outcome J: Fault in the CUT's caller.
	Outcome K: Fault in reference compiler's callee.		Outcome L: Not a possible outcome.
	Outcome M: Two conventions. One shared between the reference compiler's callee and CUT's caller, and vice versa.		Outcome N: Not a possible outcome.
	Outcome O: Not a possible outcome.		Outcome P: No faults detected.
<p>Key</p> <p>test passed →</p> <p>test failed →</p> 			

Table 4-5. All outcome configurations.

4.6.4 Test Results

We used our technique for selecting test vectors to test several compilers on several target machines. Several errors were found in C compilers on the MIPS. In this section, we present these results.

We selected several C compilers that generate code for the MIPS architecture (a DEC-Station Model 5000/125). These included the native compiler supplied by DEC, two versions of Fraser and Hanson's *lcc* compiler [FH91, FH95], several versions of GNU's *gcc* [Sta92], and a previous version of our own C compiler that used a hand-coded calling sequence generator. Although we feel that this technique is extremely valuable throughout the compiler development cycle, we believe that it would be fairest to evaluate its effectiveness in finding errors in young implementations of compilers. Where possible, we have used early versions of these compilers. These versions, called *legacy* compilers, represent younger implementations that more accurately exhibit bugs found in initial releases of compilers. However, each of these compilers is a production-quality compiler that has been widely used for years. Finding any bugs in their implementations is still a significant challenge.

In testing the compilers, we checked for two types of conformance: internal and external. Compiler A internally conforms if code that it generates for a caller can properly call code for a callee that it generated. We denote this using $A \xrightarrow{c} A$. Compiler A externally conforms if its caller code can call another compiler B 's callee code, and *vice versa* ($A \xrightarrow{c} B$ and $B \xrightarrow{c} A$). Thus, the callees and callers are compiled using each of the compilers under test. This results in n object versions for n compilers. Each caller version is then linked with the callee that was generated by the same compiler. This results in the n tests necessary to verify internal conformance for this test case. To establish external conformance, we could naively link each caller to each callee, which would yield $2n^2$ tests. However, we can do better. Recognizing that procedure call (\xrightarrow{c}) is symmetric we can easily reduce this to n^2 (since if $A \xrightarrow{c} B$, then $B \xrightarrow{c} A$). Furthermore, procedure call is also transitive, so if $A \xrightarrow{c} B$ and $B \xrightarrow{c} C$, then $A \xrightarrow{c} C$. This reduces the number to $2n - n$ as pictured in Figure 4-16. Each compiler's caller is linked to the reference compiler's callee. This facilitates the isolation of which compiler does not conform when an error is detected.

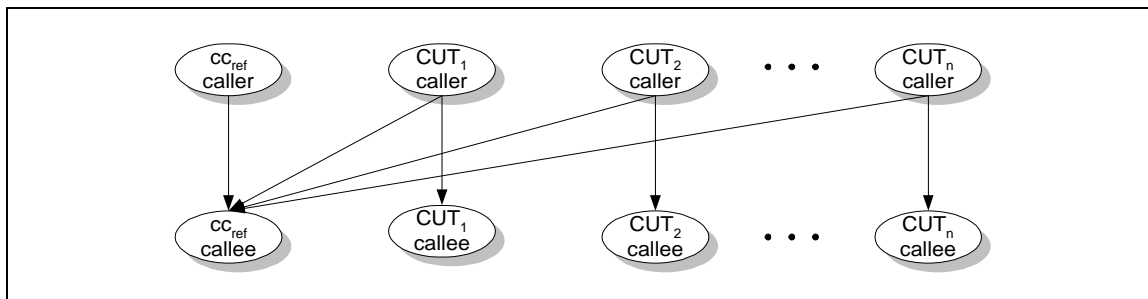


Figure 4-16. Determining conformance of n compilers.

The results of running both internal and external tests on the compiler set for the MIPS are shown in Table 4-6. We found both internal and external conformance errors in all of the tested compilers. Table 4-6 reports internal and external errors separately. Within each class, the number of actual tests that failed and the number of faults that caused failure are indicated¹. The numbers reported in the fault columns indicate the approximate number of actual coding errors resulting in test failures. These numbers are only approximate. We tried, as best we could, to glean this information from the results of tests. More accurate numbers can only be obtained by examining the compiler's source.

Compiler	Internal		External	
	Failed Tests	Faults	Failed Tests	Faults
cc (native)	2,346	1	2,346	1
gcc (1.38)	2,370	2	2,567	3
gcc (2.1)	0	0	2,346	1
gcc (2.4.5)	1	1	2,374	3
lcc (1.9) ^a	0	0	0	0
lcc (3.3)	2,407	2	2,407	2
vpcc/vpo	2,346	1	486	3
Total	9,470	7	12,526	13

Table 4-6. Results of running the MIPS test suite on several compilers.

- a. Version 1.9 of lcc was not tested using *varargs* because we could not get the compiler to accept *varargs* callees. This could either be a problem with the compiler, or the particular version of `stdarg.h` on our machine.

1. These numbers include tests of both standard procedure calls and variadic procedure calls.

4.6.4.1 Standard Procedure Calls

Internal conformance errors were found in two versions of *gcc*. *gcc* 1.38 failed 24 tests that focus on passing structures in registers. Structures between nine and 12 bytes in size (three words) are not properly passed starting in the second argument register. Procedure signatures that correspond to these tests include:

```
void(int, struct(9-12));
```

gcc 2.4.5 fails a single test. The fault occurs with procedures with the signature:

```
void (struct(1), struct(1), struct(1));
```

gcc 2.4.5 fails to even compile a procedure with this signature¹. The fact that *gcc* 2.1 does not have this error indicates that the error was *introduced* after version 2.1. This supports our conjecture that such method of automatic testing is extremely useful throughout the development and maintenance life-cycle of a compiler.

External conformance errors were more prevalent. *gcc* 1.38 does not properly pass 1-byte structures in registers. This results in 208 test case failures. *gcc* 1.38 and 2.4.5 cannot pass a structure in the third argument register when that structure is followed by another. The fault occurs with signatures matching:

```
void(int, int, struct(1-4), struct(any));
```

This results in another 13 test failures. Finally, *vpcc/vpo* has 486 tests that fail. Two faults are responsible: 1) structures are not passed properly in registers, and 2) 1 to 4-byte structures are not passed in memory correctly if they are immediately followed by another structure. These match signatures:

```
void (int, int, int, int, struct(1-4), struct);
```

4.6.4.2 Variadic Procedure Calls

Procedures that take variable-length argument lists (variadic functions) are written using one the of two standard header files: `varargs.h` (for traditional C) and `stdarg.h` (for ANSI C). These files provide a standard interface for the programmer to write variadic functions. Because a variadic function's caller uses the standard procedure calling convention, the variadic callee must also conform to this convention. The following paragraphs detail the results of calling callees that are implemented using *varargs/stdarg*.

1. The error returned by *gcc* 2.4.5 was:

```
gcc: Internal compiler error: program ccl got fatal signal 4.
```

Most variadic functions in C have signatures similar to the standard library function `printf`:

```
void func(char *, ...);
```

The function determines the number of arguments from the first parameter. However, functions of the form:

```
void func(double, ...);
```

are also valid. When running test cases that contained variadic functions whose first argument was a `double`, we found that none of the compilers, including the reference compiler, properly implemented the calling convention. The difficulty stems from the fact that until the type of the argument is known, the callee cannot determine whether to fetch the first argument from the floating-point register or the integer register. Most implementations of *varargs* dump the contents of the argument-passing registers to the stack in the function's prologue. For calling conventions like the MIPS, a more sophisticated solution must be used. This error caused 2,346 test cases to fail for all of the compilers. Version 2 releases of *gcc* managed to avoid this problem at the expense of interoperability; their generated callees do not conform to the established calling convention.

From these results, obviously the state-of-the-art in compiler testing is inadequate. Because these are production-quality compilers, each of them has undoubtedly undergone rigorous testing. However, hand development of test suites is an arduous and itself error-prone task. Furthermore, because these tests are target specific, they must be revisited with each retargeting of the compiler. In contrast, by using automatic test generators that are target sensitive, compilers can quickly be validated before each release.

4.7 Summary

Current methods of procedure calling convention specification are frequently imprecise, incomplete, or contradictory. This comes from the lack of a formal model, or specification language that can guarantee completeness and consistency properties. We have presented a formal model, called P-FSA's, for procedure calling conventions that can ensure these properties. Furthermore, we have developed a language and interpreter for the specification of procedure calling conventions. With the interpreter, a P-FSA that models a convention can be automatically constructed from the convention's specification. During construction, the con-

vention can be analyzed to determine if it is complete and consistent. The resulting P-FSA can then be directly used as an implementation of the convention in an application.

Although we have shown that it is possible to automatically generate the calling sequence generator of a compiler, some work is required to retrofit an existing compilation system to use CCL descriptions. Fortunately, it is possible to reap the benefits of CCL without any modification of the compiler. Using automated compiler tools and testing, one can significantly increase the robustness of *any* compiler. We have combined these two techniques, in a new way, that further closes the gap between actual compiler implementations and the ever-sought-after correct compiler. By using formal specifications of procedure calling conventions, we have designed and implemented a technique that automatically identifies boundary test cases for calling sequence generators. We then applied this technique to measure the conformance of a number of production-quality compilers for the MIPS. This system identified a total of at least 22 faults in the tested compilers. These errors were significant enough to cause over 2,300 different test cases to fail. Clearly, this technique is effective at exposing and isolating faults in calling sequence generators of mature compilers. Undoubtedly, it would be even more effective during the initial development of a compilation system.

CHAPTER 5

COMPUTING SYSTEM DESCRIPTION LANGUAGE

Chapters three and four presented two description languages that describe distinctly different machine features. In this chapter, we show how to we bring these two descriptions together using a general framework for building reusable descriptions for systems software.

For years, machine descriptions have been used in a variety of ways to parameterize software implementations. However, except in the very rarest of instances, none of these description systems have been reused. It is clear that for any new description language to make a contribution to the state-of-the-art, it must bring more than just the ability to describe a single machine feature for a single application. Thus, we put forth the following observations about description systems and their use:

- Application dependence stifles sharing. If a description is tailored to a particular application, it will be difficult for a new application to make use of the description.
 - Application dependence is inevitable. No matter how pure the intention, or what form the description takes, machine dependence will always creep into descriptions. Often, modification of the description to suit the application is easier than modifying the application to suit the description.
 - Partial descriptions are useful. Many applications only require a subset of information. For example, many instructions are never generated by compilers. If the application does not use the information, the application writer should not be required to provide the information in the description.
-

- Comprehensive descriptions are large. Modern computing systems are complex and applications often view the same information in different ways. Capturing all of these details will cause descriptions to become large and monolithic. Organizational tools should be available.
- Writing new descriptions is difficult. Often, just gathering the information to be described can be a significant obstacle.
- Different applications may view the same machine features differently. Compilers often view instructions in assembly language format, whereas simulators may view instructions in their binary format.
- Descriptions and their languages are continuously evolving. After a description is initially written, it is likely to live in constant flux as new machine models are introduced, as additional machine features are included, and as new applications make use of its content.

Obviously, modern description systems must contend with considerable demands, many of which appear to be contradictory. After consideration of these observations, we propose the following set of design goals:

1. Application independence should be encouraged.
2. Application dependence should be tolerated.
3. Partial descriptions should be permitted.
4. Descriptions should be modular and composed from simple components.
5. Descriptions should be reusable.
6. Different views of the same feature should be permitted.
7. Descriptions should be extensible.

Any description system should strive to meet each of these goals—particularly application independence. In this chapter, we present a system for building computing system descriptions that are application dependent, but can still be shared among many applications.

5.1 CSDL Overview

Our system extends previous work in machine descriptions in four key ways: abstraction level, extensibility, reusability, and modularity. Because this new description system widens the abstraction level of machine descriptions, we call them *computing system descriptions* to reflect

their broader applicability [BD96a]. 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.

5.1.1 Modules

As shown in Figure 5-1, 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 described in chapter three. 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 nearly 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.

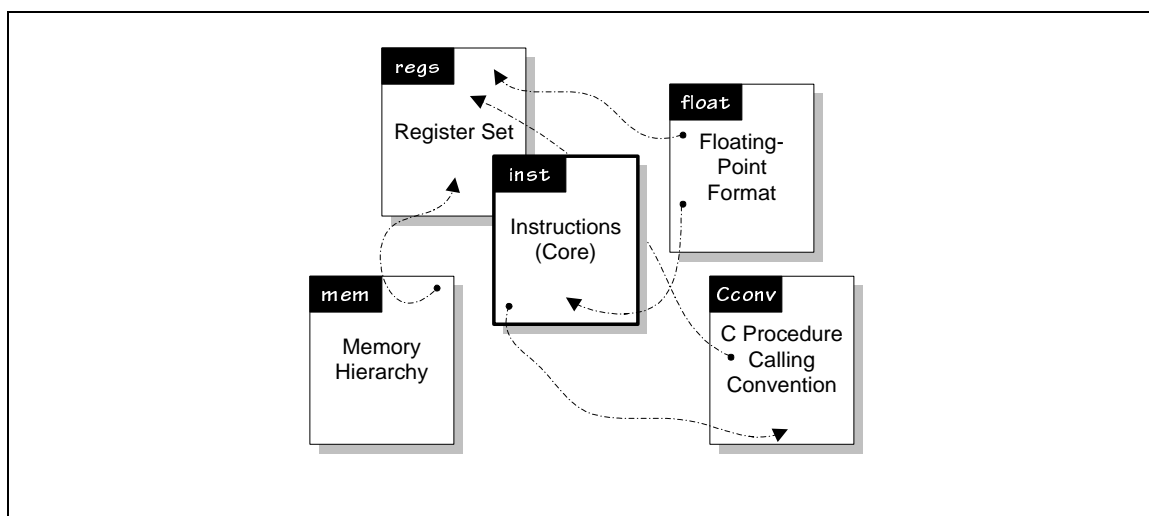


Figure 5-1. Computing system description framework.

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 the part of the description that involves 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.

Because CSDL descriptions are modular, significant flexibility is available to each application. The disadvantage of dividing 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: objects, linked values, application annotations, and object aspects. These mechanisms are the *glue* that holds CSDL descriptions together, and give them their descriptive power.

5.1.2 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. 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 throughout the system to the other modules using links.

Figure 5-2 demonstrates module linking. A register description excerpt, shown in Figure 5-2(b), 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 5-2(a), contains references to these two values. To accurately define the valid instructions for the machine, the instruction description must know what register indices are valid. The instruction description refers to the valid register indices by name. Changes to the register description are immediately reflected in each referencing module.

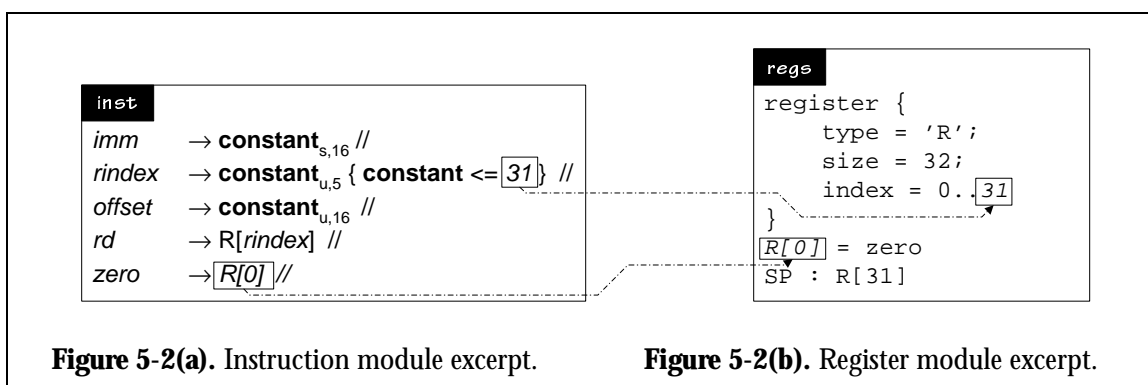


Figure 5-2. Linked values.

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. They also 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.

5.1.3 Application Annotations

The primary shortcoming of previous machine description techniques is that they present information in an application-dependent way. While the inclusion of application-specific information makes the descriptions easier for the particular application to use, it frequently makes the descriptions useless for other purposes. CSDL provides *application annotations* to reconcile these differences.

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 appli-

cation is viewing the description, the annotations appear as part of it, whereas when other applications view the description, the annotations are not present. Annotations can be thought of as an overlay, as shown in Figure 5-3, which an application places over a module. The application developers can include whatever information they wish without impacting other applications that are using the same module.

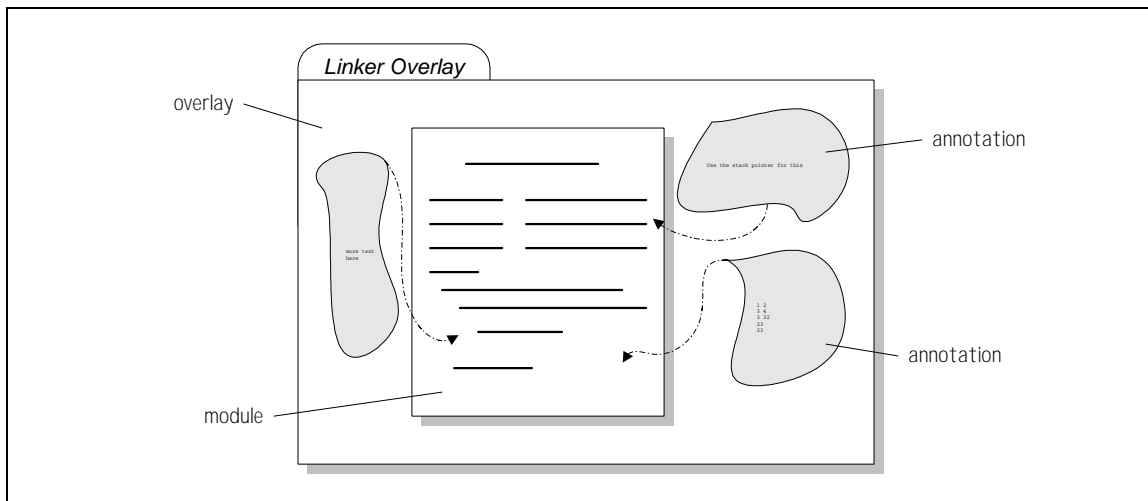


Figure 5-3. An application's annotation overlay.

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 τ RTL instruction description is pure¹, 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 this information from the description, an annotation can be attached to the OR instruction, as shown in Figure 5-4, to indicate that a specific form may be used to achieve the move.

1. 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 misled.

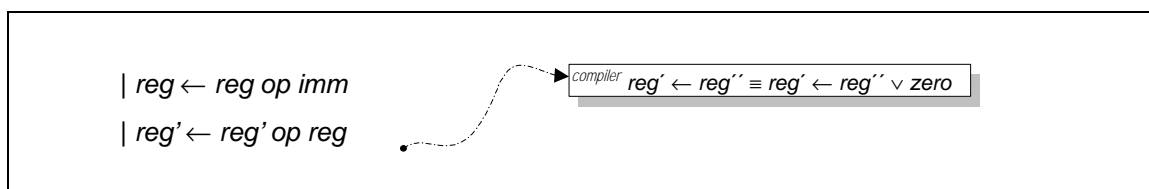


Figure 5-4. A CSDL annotation.

5.1.4 Module Aspects

A concept closely related to annotations are module aspects. Although 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 feature 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 may be made 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" and placed over the module. This provides a mechanism for components to have many facets that are used by many applications.

Figure 5-5 illustrates the use of aspects. Here, the core description is augmented with two aspects: an assembly language aspect and a binary format aspect. Although aspects are usually keyed using color, here they appear as labeled boxes. The assembly aspect is shown in the left box, while the binary format aspect appears on the right. 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.

<i>reg</i> → « R[<i>rindex</i>]	assembly <i>\$rindex</i>	binary <i>rindex</i> »//
<i>zero</i> → « R[0]	assembly 0	binary 0 » //
<i>op</i> → « +	assembly <i>addi</i>	binary <i>ADDI</i> »
« -	assembly <i>subi</i>	binary <i>SUBI</i> » //
<i>arith</i> → « <i>reg'</i> ← <i>reg''</i> <i>op</i> <i>imm</i>	assembly <i>op reg', reg'', imm</i>	binary [<i>op, reg'', reg', imm</i>] »
« <i>reg'</i> ← <i>reg''</i> <i>op1</i> <i>reg''</i>	assembly <i>op1 reg', reg'', reg''</i>	binary [<i>SP, reg'', reg'', reg', 0, op1</i>] »
//		

Figure 5-5. Assembly language and binary format aspects of instructions.

5.2 Module Processing

Each CSDL module describes some machine feature. In each module, one language must be designated as the *host*. The language provides the skeleton to which CSDL aspects and annotations are attached. Typically, the host language defines the kinds of objects, and indirectly, the level of abstraction for the module. In contrast, the languages used in aspects and annotations are called *guest* languages. Guest languages are typically small languages that augment the information provided by the host language. However, their syntax is not dictated by either the host language or CSDL.

5.2.1 CSDL Language Processing

The CSDL language processor must contend with elements from two or more languages: CSDL (objects, annotations, *etc.*), the host language, and possibly embedded guest language elements embedded in CSDL annotations and aspects. CSDL provides all necessary facilities to process and separate each language's elements while maintaining the semantic linkages between the language elements. This process is achieved by providing CSDL with language processors (scanners and parsers) for the host language and each guest language. CSDL reads the CSDL source and dispatches strings of symbols to the appropriate processor as shown in Figure 5-6.

A grammar for the abstract syntax of CSDL modules is shown in Figure 5-7. We use a standard grammar syntax augmented with regular expression syntax. Terminals are shown in bold, nonterminals are shown in *italic*, brackets ('[' and ']') indicate optional (zero or one) instances, and Kleene star ('*') indicates zero or more instances of a grammar symbol. A name token is a string of one or more alphabetic characters and a string token is a string of one or more symbols. From the grammar, it is clear that CSDL imposes little structure on the

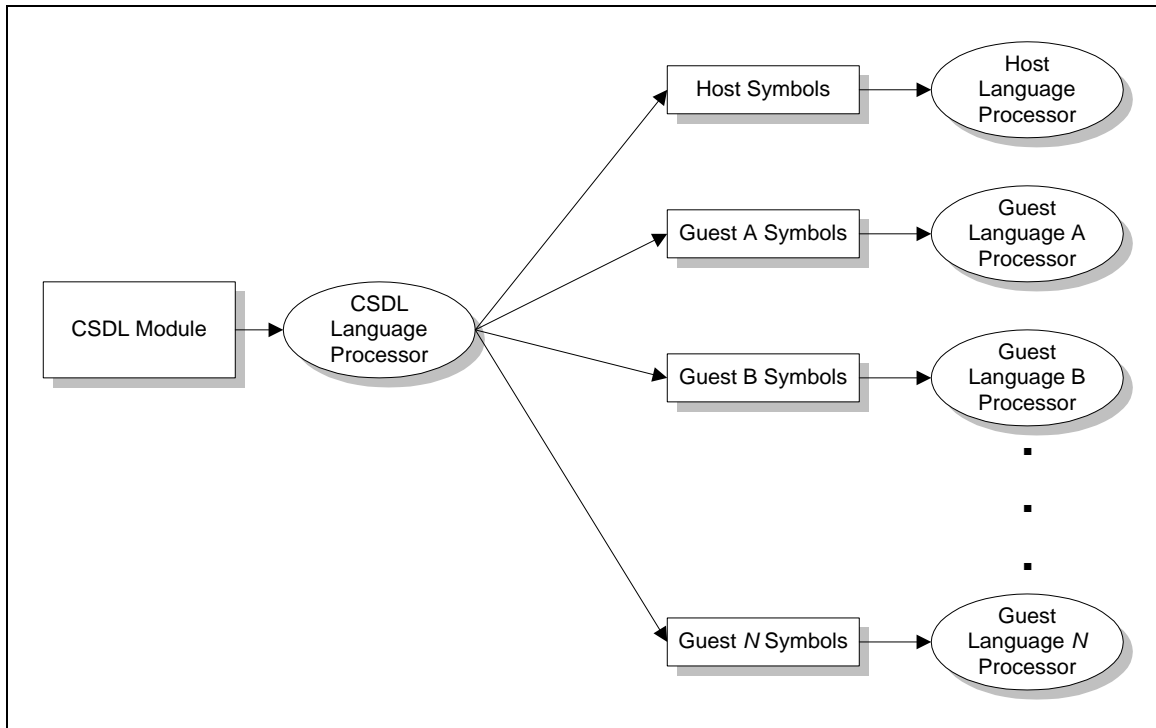


Figure 5-6. CSDL Language Dispatching.

embedded languages. A CSDL module is simply a string of host language tokens and CSDL language elements (except aspects). An object is a string of host language tokens and CSDL language elements. Annotations and aspects are strings of guest language tokens and CSDL language elements. Aspects may only be found in CSDL objects. A link is simply a cross reference to a CSDL object definition, and as such, may produce any string an object can contain.

1.	<i>module</i>	→ (module [name] <i>mvalue</i> *)
2.	<i>mvalue</i>	→ <i>object</i> <i>string</i> <i>annotation</i> <i>link</i>
3.	<i>value</i>	→ <i>mvalue</i> <i>aspect</i>
4.	<i>object</i>	→ (object [name] <i>value</i> *)
5.	<i>annotation</i>	→ (annotation name <i>value</i> *)
6.	<i>link</i>	→ (link (name+ [(name)]) <i>value</i> *)
7.	<i>aspect</i>	→ (aspect name <i>value</i> *)

Figure 5-7. CSDL Grammar.

Obviously parsing CSDL modules is simple enough. However, the string token can contain either host language elements or any guest language elements. Using the grammar, it is easy for CSDL to identify which language the string belongs to, but it is impossible, without more information, for CSDL to identify if the string contains only valid language tokens

or if the string of language tokens is in the given language. Take, for example, the following abstract CSDL string:

(*module name* **(** *object name* *htoken htoken ... htoken* **(** *aspect name* *gtoken gtoken ... gtoken* **)** *htoken htoken ... htoken* **)** *)* **)**

Processed by CSDL
Processed by host language
CSDL
Processed by guest language
host

CSDL terminals are shown in bold. Non-terminals for all languages are shown in italics. Each string of tokens is labeled by the language to which it belongs. In this example, a sequence of host language tokens has been named using the CSDL naming mechanism. This creates an object which, in turn, may have aspects attached to it.

The procedure for processing CSDL strings is best illustrated with a concrete example. The following excerpt from Figure 5-5:

CSDL
CSDL
CSDL
CSDL

arith → « *reg' ← reg'' op imm* *assembly* *op reg', reg'', imm* *binary* *[op, reg'', reg', imm]* »

Core Host
Core Module Host Language
assembly Guest Language
binary Guest Language

describes a single production from a CSDL core grammar. In this example, the module contains language elements from four languages: CSDL, the core host language, and two guest languages. In order for CSDL to properly process this string, it must have a way to identify host and guest tokens. To demonstrate why, we examine what must happen to successfully process the above example.

Our excerpt picks up during core language processing. When the CSDL begin object token ('«') is encountered, we must mark the beginning of the CSDL object that is embedded in the core language string. When we reach the assembly aspect, we must switch processing from core language scanning to assembly aspect scanning. Similarly, when we reach the binary aspect, we must switch to the binary aspect scanner. Finally, when we encounter the CSDL close object token ('»'), we must complete the object definition by associating the results of processing the aspects with the designated core substring. To enable this parsing, host and guest languages must provide token and language information in the form of Lex [LS83] and Yacc [Joh83] specifications. When processing a module, the CSDL parser absorbs all CSDL tokens. These direct the processor to switch processing between CSDL, the host language, and the guest languages. This process is illustrated in Figure 5-8.

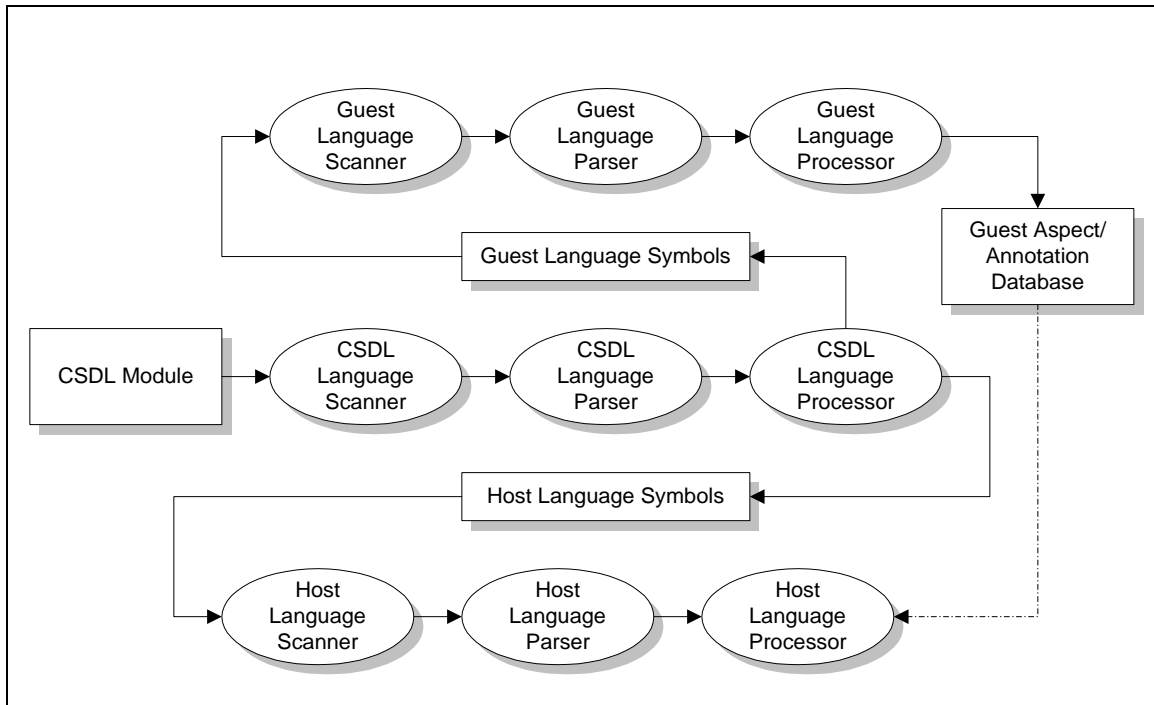


Figure 5-8. Processing of a CSDL module.

First, all tokens are scanned and parsed by the CSDL front end. Based on the CSDL tokens, the CSDL processor passes symbols that are not part of CSDL tokens to either the host language scanner or a guest language scanner. These are, in turn, passed to their respective parsers and processors.

When the CSDL processor encounters the following tokens, it takes the indicated action:

- **module** — this is the beginning of a CSDL module.
Action: the host language is set as the current language.
- **string** — this string of symbols belongs to the current language.
Action: the string is passed symbol-by-symbol to the current language's scanner. Beginning and end positions of the language's tokens are noted (processing may not end in the middle of a token).
- **aspect, annotation** — this marks the beginning of a guest language string of symbols.
Action: push the current language on a language stack. Set the indicated guest language as the current language.

- **object** — the enclosed string of symbols is a CSDL object.
Action: identify the current language's grammar production that derives the enclosed string of symbols. The current language is unchanged.
- **link** — the name refers to a named CSDL object.
Action: the derivation of some production in the current language is found within the link. This derivation is passed, symbol-by-symbol to the current language's scanner.
- **)** — that matches a token that switched the current language (an **annotation** or **aspect**).
Action: the current language is switched back to the previous language by popping it from the language stack.

By processing the module in this way, it is possible to identify which symbols belong to which languages. Because CSDL embeds strings of host and guest language tokens, CSDL must ensure that strings of symbols derive complete strings of tokens in the given language (aspects and annotations cannot begin or end in the middle of a language's token, nor can they be embedded in the middle of a token from the enclosing language). In the case of guest language strings, CSDL must also ensure that complete strings of guest language tokens are derived from the guest language's grammar start symbol.

In order to satisfy the above requirements, CSDL must be aware of the processing of symbols that each language performs. We achieve this by modifying the Lex and Yacc routines for each of the languages. Lex and Yacc produce their respective scanners and parsers by producing tables from their input specifications. In the case of Yacc, these tables are used to direct the shift and reduce actions of a general purpose parser. For Lex, they direct the tokenizing actions of a general purpose scanner. So, a parser is built by linking a table specifying the shift and reduce actions for the given language with a library of routines, called the parser skeleton, that actually perform the parsing based on the table. We replaced the stock Lex and Yacc skeletons with custom skeletons that interact with the CSDL language processor. These routines are used to redirect the scanner's input to strings provided by CSDL and provide information to CSDL regarding the state of both the scanner and parser for a given language. This is why we require that guest and host language processor use Lex and Yacc specifications. However, the same technique could be used with any parser or scanner generator system. Handwritten parsers or scanners would, of course, have to provide these linkages to CSDL.

Obviously, the processing of CSDL modules is tightly integrated with the processing of each of the embedded languages. However, as long as Lex and Yacc specifications are provided, the host and guest language designers need not concern themselves with the tight interaction between CSDL and the language syntax analysis. Furthermore, producing CSDL modules that embed such a variety of languages might, at first, seem difficult. Fortunately, the environment that is used to edit these modules provides the necessary support to make the writing of even complex modules manageable. In the next section, we present the CSDL editing environment.

5.2.2 An Environment for CSDL

CSDL provides a flexible system for embedding guest languages that extend module host languages. In addition, host and guest languages have available to them facilities for using extended character sets (*e.g.*, Greek alphabetic symbols), advanced character formatting using font variations (*e.g.*, bold, italic, *etc.*), and character positioning (*e.g.*, subscripts and subscripts). Even with the most advanced text editors, such as emacs, these facilities are just becoming available. Instead, we turned to Adobe's FrameMaker desktop publishing system [Ado97a]. FrameMaker provides all of these editing features to the user natively while it also provides two ways to extend the FrameMaker application: the FDK (Frame Developer's Kit) [Ado97b] and MIF (Maker Interchange Format) [Ado97c]. We use both of these to access the CSDL module source code written in FrameMaker.

Because FrameMaker was designed for formatting large documents, it provides a number of features that support the creation and maintenance of large descriptions as well. These include hypertext links, text variables, text inclusion from other documents, cross references, and conditional text. Currently, CSDL makes use of but a few of these features. After more experience, we may leverage off more of the features that FrameMaker provides.

5.2.2.1 Supporting Annotations and Aspects

The two most visible features of CSDL are annotations and aspects. Both of these features need significant support from the editing environment. Since these features allow embedding of languages within the CSDL module, at a minimum there need to be ways of delimiting the embedded language's text and naming which language the text belongs to. Further, it is desir-

able to easily select which annotations and aspects to view or process. FrameMaker's conditional text feature has these capabilities.

In FrameMaker, regions of text can be associated with one or more "conditions." In turn, these conditions have two properties: visibility and display characteristics. The condition name and its visibility property are "out of band" data. That is, they are not visible in the document itself. However, the display characteristics do control how the conditional text is displayed. Typical display characteristics include underlining, bold, italic, and font color. These font variations cue the reader that the text belongs to a particular condition.

We use conditional text to mark both aspects and annotations. Typically, different font colors are used to indicate different aspects or annotations. For example, in instruction descriptions, an assembly language syntax aspect could be displayed in red, while the instruction's cost could be displayed in blue. With each instance of conditional text, the name of the condition is available for CSDL to examine. Thus, while FrameMaker and the user think of aspects as colored text, CSDL just views the text as being associated with a particular aspect name or identifier. In addition, when the user is concentrating on one aspect of a description, such as assembly language syntax, they can disable the display of all of the other aspects by toggling individual visibility properties. So, while a particular CSDL object might have a plethora of different aspects, the author can selectively view those aspects that require inspection or modification. CSDL can, of course, do this as well; this yields a powerful technique for constructing custom descriptions from general descriptions.

Using conditional text also makes descriptions more compact. Neither the aspect or annotation's name, nor its beginning and ending delimiters take up space in the text of the description. This comes at a cost though: printed descriptions are not necessarily complete (because not all of the conditional text tags are visible) and the names of aspects and annotations are not printed on the page. Further, if one chooses color as the distinguishing display property, and the description is printed on a monotone printer, the display characteristics are lost as well. However, we feel that the advantages that conditional text provides in building manageable descriptions outweighs these printing deficiencies.

5.2.2.2 Extended Character Sets and Token Matching

By using a desktop publishing system, CSDL presents new issues in language design. When is it appropriate to use special characters? How does the addition of font variation impact scan-

ning? How can different fonts and positioning be effectively used in language design? Each of these questions deserves more attention than we can give them here, so we will only summarize our limited experience with these issues in this section.

Using special characters can deliver great semantic benefits. Long have ASCII-based language designers struggled with the shortage of special symbols. For example, the Boolean operators are typically synthesized from multiple symbol sequences, such as ‘&&’ for AND and ‘||’ for OR in the C programming language, or previously unused symbols that yield expressions that are not intuitive, such as the exclusive OR (^) operator in C. Having to access to the symbols ‘^’, ‘v’, and ‘⊕’ that are traditionally used for these operations in Boolean expressions greatly increases the readability of languages that use them.

Nevertheless, the addition of special characters can be abused. Take for example, the use of ‘⊥’ in the CCL language. This symbol is used as a selection operator. However, there is no historical use of this symbol for this purpose. Simply using a special symbol for no other reason than its availability (or because it makes expressions more concise) is not appropriate. We must be sure to make judicious use of special symbols if we hope to increase the readability of languages rather than to further obfuscate the notation as in the tradition of the C exclusive OR operator.

In addition to special symbols, using a modern desktop publishing system gives the language designer access to font variation in token specification. This is not, by far, a new concept. Programmers of case-sensitive languages have used capitalization of identifiers for years to impart the semantics of scope (capitalized for globals, lowercase for locals) and type (all caps for macros, capitalization conventions for functions, *etc.*). With the addition of font variation, such as bold and italic, language designers can separate the namespaces of keywords and identifiers. For example, as is often done with program pretty printers, bold can be used to indicate language keywords. In τ RTL descriptions, we use italic to distinguish between grammar terminals and nonterminals. We also use character positioning (superscripting and subscripting) to build expressions that, in traditional computer languages, would use a bracketed notation. For instance, τ RTLs use superscripting for bit selection: $expr^{m..n}$. If character positioning were not available, two brackets would be probably be used to delimit the beginning and end of the superscripted expression. By using character positioning, we achieve a more concise expression without sacrificing readability.

Clearly, there are some situations in which special symbols and font formatting can be effectively used in language design. This does, however, come at a cost in implementation complexity. For each character of a source file, the font and variations must be recorded to distinguish between visually different instances of the same character ('⊥' and '^' are the same character in different font families, and 'X' and 'X' are the same character with different font variations). These differences cannot be encoded in the same byte as the character's value, so at least an additional byte must be used to encode the character's format. Therefore, languages that use special symbols and font variation must be capable of reading and matching multi-byte characters.

An interesting problem with building scanners that handle multi-byte characters is the difficulty of recognizing tokens with font variations. For example, in τ RTL where we use subscripting to designate value interpretation, we might encounter the following two different token strings: $\text{expr}_{b,32}$ and $\text{expr}_{b,32}$. These two expressions probably do not *appear* differently to the author, but the scanner will see that the subscripted comma (','), is, in fact, italicized in the second expression. We would like to think of these two sequences of tokens as equivalent since either version might occur when writing the expression. In this case, there is really no semantic difference between italicizing the comma and not. In contrast, we want to think of the two token strings $\text{expr}_{b,32}$ and $\text{expr}_{b,32}$ as different because the 'b' is italicized in the second expression but not in the first, indicating that 'b' is a τ RTL variable rather than a terminal in the language. In this case, the meaning of the two expressions, as defined by the language, is dramatically different. It is the job of the token scanner to match either version of the comma to the same comma token and to match the two instances of the 'b' token to different tokens.

To achieve the desired result, the scanners for languages that make use of enhanced character formatting match symbols in the following manner. Each symbol that makes up a token is described using two bytes. The first indicates which character in the font is to be used. The second encodes eight bits of font formatting information. As the characters are extracted from the FrameMaker document, these bytes are generated by recording current font settings for each character. Four bits are used for font name (or number) and the other four are used to indicate bold, italic, superscript, and subscript. The tokens in the Lex scanner specify the values of each of these formatting bits. For each bit, the token may specify "set," "not set," or "don't care." In our above examples, the "variable" token specifies the italic bit as

“set” for each of its characters while the “comma” token specifies all bits as “don’t care” since none of the variations, including font, influence its identification as the comma token.

The correct specification of tokens that include font formatting requires careful attention to detail. If tokens are specified too loosely, multiple token definitions will overlap¹. If tokens are specified too tightly, numerous tokens that appear valid to the user will be rejected by the scanner. This behavior yields a language that is frustrating for authors to use. This interaction is most apparent with subtle changes in font variation such as italic in combination with small symbols (punctuation), or superscripting, or subscripting. There is probably no instance in which period ‘.’ should set formatting bits to anything but “don’t care!”

5.2.2.3 Objects

CSDL provides minimal support for objects. An object is defined to be any sequence of tokens that is derived from a language’s grammar production. Put another way, objects are used to name instances of grammar production derivations. This is about as concrete a definition of an object as CSDL can provide given that language grammars and token specifications are the only things that CSDL knows about host and guest languages.

A CSDL object is delimited using french quotation marks (guillemet): ‘«’ ‘»’. These symbols were chosen because they are intuitively delimiters that are not already available to ASCII based languages. So, although we preclude future languages from using these symbols, we can be fairly certain that no existing language uses them. Using these symbols, we can mark the beginning and end of a CSDL object, as in the τ RTL line from the DLX instruction description:

$$addr \rightarrow \ll (\text{reg}_{u,32} + \Delta(\text{constant}_{u,16})_{u,32})_{u,32} \gg$$

Here, we have designated the entire right-hand-side of the rule as a CSDL object. The object corresponds to a derivation of a grammar symbol from the τ RTL core description language’s grammar (this is a grammar whose start symbol derives grammars). This is the grammar symbol that describes what may be on the right-hand-side of a τ RTL grammar. Conceptually, the above example is one of several addressing modes on the DLX. Extending this example, we

1. Luckily, this overlap can be identified by Lex automatically. If this were not the case, whole classes of tokens would never be matched by the scanner.

can attach an assembly language aspect that describes the syntax of the assembly language for this addressing mode:

$$addr \rightarrow \ll (reg'_{u,32} + \Delta(\mathbf{constant}'_{u,16})_{u,32})_{u,32} \overset{\text{assembly}}{\mathbf{constant}'(reg')} \gg$$

We use a box here to indicate the beginning and end points of the aspect. However, usually the aspect would be indicated using a different text color from the surrounding text. The underlined text corresponds to a guest language that describes the assembly language syntax. As often is the case, the guest language depends on the syntax of the host language. Here, the variables in the host language derive τ RTL expressions, which also have assembly language aspects associated with them. Thus, we have two parallel derivations: one that derives the τ RTL expression while the other—the one found in the aspects—derives the equivalent assembly language expression. When processing the τ RTL grammar, the host language would then have access to the associated assembly language aspect.

5.2.3 Processing Summary

CSDL provides a rich set of features that support modular description development and description extensibility through language embedding. Language elements from different languages are distinguished by the color of their text. Symbols that are not available on standard computer keyboards may be used to enhance the readability of the embedded languages. Font variations can be used to strengthen the recognition of language elements. The description editing environment, although not a programmer's editor, is as familiar environment: the desktop publishing system.

Once descriptions have been written in FrameMaker, CSDL extracts the text using the Frame Developer's Kit that provides an API to the underlying FrameMaker document. The result is a CSDL module that contains CSDL directives that mark the beginning and end of CSDL objects, aspects, annotations, and links. Host and guest language symbols are represented using two-byte pairs that encode the character and various font display characteristics. As the CSDL module is processed, symbols are passed to the appropriate host or guest language scanner for tokenizing. The scanner must consider font formatting when trying to determine which token type to match. Tokens are passed up to the language's parser, where parser actions are recorded by the underlying CSDL processor. CSDL objects always name entire language grammar production derivations. CSDL associates each object's aspects with

the attribute that the parser pushes onto the parsing stack upon the grammar rule reduction that represents the object's derivation production. Using this mechanism, it is possible for the host language processor to gain access to the guest language aspects.

5.3 Applications

CSDL has been designed as a multi-application description framework from the outset. Applications can use or extend existing CSDL modules, or add additional modules to suit their needs. In this section, we present a couple of examples of how different systems software applications could use CSDL to build descriptions that can be shared among many applications.

5.3.1 Binary Translation

Binary translators take executable programs for a source machine and translate them to executable programs for a target machine. This application requires information about the binary instruction format for two machines [AKS00, ZT00, GAS+00, CE00]. At first, it may seem that an implementation could simply take two instruction descriptions and automatically derive the translation from one format to the other. However, binary translation is not such a simple problem [CER99]. Often, the necessary translation from one instruction set to another is not readily apparent; human intervention is necessary to glean the proper translation.

Consider a translator that converts SPARC executables into MIPS executables. An alternative approach to the one described above would be to annotate the SPARC description with the necessary information to perform the translation to MIPS instructions. This can be accomplished using a CSDL SPARC description by adding a MIPS translation aspect to each of the SPARC instructions. Figure 5-9 shows an excerpt from such solution.

To each SPARC instruction, a piece of C code is attached to perform the necessary translation. For example, the SPARC contains a load instruction that uses an indexed addressing mode. Since the MIPS doesn't contain an indexed mode, one is synthesized using two instructions: an add and a load word instruction. The MIPS and SPARC also differ on the size of their immediate operands. On the SPARC, an immediate value is 13 bits, while on the MIPS, an immediate value is 16 bits. Thus, SPARC load-immediate instructions are trivial to

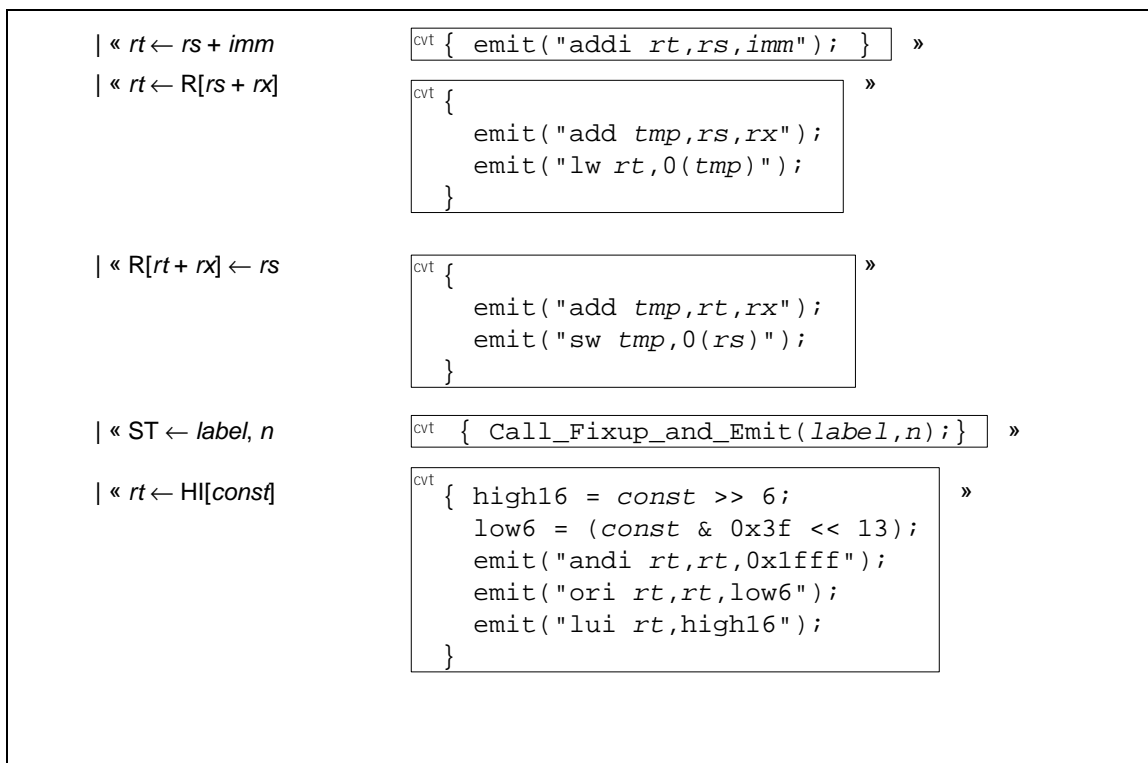


Figure 5-9. Specifying binary translation using a CSDL aspect.

translate to MIPS instructions since SPARC immediate values are always smaller. However, to form a 32-bit constant, the SPARC has a `sethi` instruction that loads the high 22 bits of a register. When such an instruction is encountered, several MIPS instructions must be emitted to synthesize the load.

It is possible that one could automatically generate the translations described above. However, for complex instructions, such as `CALL`, it is unlikely that an automatic process will succeed. Such instructions assume other processor state, such as a particular stack layout or register usage. However, by attaching small portions of C code, we can easily reference external handwritten functions that perform these complex translations. Other situations, such as operating system traps, exception handlers, and instructions that use special-purpose registers can be handled in a similar way.

By providing extensions to modules, CSDL permits applications to embed application-dependent information into otherwise application-independent descriptions. In our example, the translations are application dependent, however, the instruction description is not. Although the description contains the binary translator's translation methods, CSDL's

aspect mechanism makes it easy for other applications to filter out these application-specific instruction aspects and make use of the rest of the instruction description.

5.3.2 Specifying a Procedural Interface to Assembly Language

Often, machine descriptions are used to map the description's notation (τ RTL in our case) that is known to the application to some format (*e.g.*, assembly language, binary instruction formats, *etc.*) whose format or notation is foreign to the application. This would occur, for instance, in the implementation of retargetable code generator. However, in some circumstances, the opposite is desired: a mapping *to* the description's notation *from* some other notation. A link-time optimizer that reads binary instructions and manipulates them in τ RTL would perform such a translation. In this section, we briefly discuss how translation in this direction can be achieved.

The New Jersey Machine-Code Toolkit [RF95] includes a Specification Language for Encoding and Decoding (SLED) [RF97] machine language (binary) instructions. From SLED specifications, one can generate interfaces that can encode or decode machine language instructions. A SLED interface is just a proceduralized assembly language. For example, a SLED assembly language interface for the MIPS would include the following C language functions:

```
Addr addr(int imm, unsigned rs);
void add(unsigned rs1, unsigned rs2, unsigned rd);
void sub(unsigned rs1, unsigned rs2, unsigned rd);
void sw(unsigned rt, Addr addr);
void lw(Addr addr, unsigned rt);
```

Using such an interface, the New Jersey Machine-Code Toolkit can generate either binary or assembly language instructions. What if, instead, you wanted to generate τ RTL? The answer is that this is easily accomplished by providing an identical interface that generates τ RTL instead of assembly language.

Figure 5-10 shows a brief excerpt from the MIPS τ RTL description that has been decorated with SLED aspects. The right-hand sides of τ RTL grammar productions are CSDL objects, each of which has a SLED aspect. The SLED aspect specifies the C language signature for the function that will emit τ RTL trees. As before, these aspects build a grammar parallel to the τ RTL grammar on the left. To build the necessary functions, we enumerate all of the C

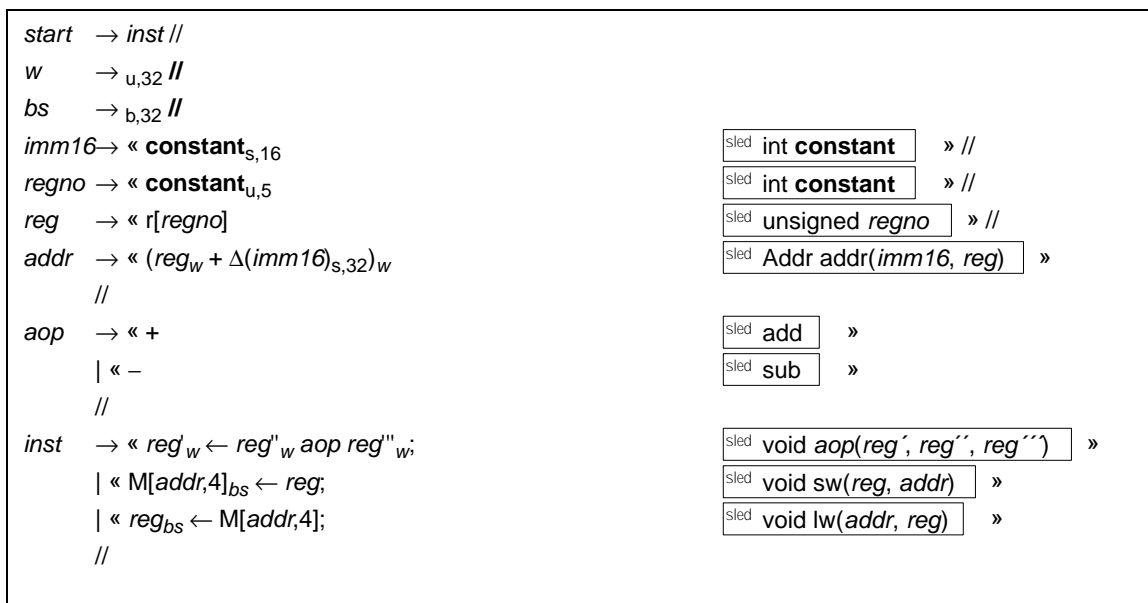


Figure 5-10. A small MIPS excerpt with SLED aspects.

signatures that can be derived from the grammar start symbol discarding duplicates when they are produced. As each signature is generated, the matching τ RTL tree is constructed. From the signature and the tree, a C function that matches the signature can be generated that, itself, generates the matching τ RTL tree.

Just as we can use this technique to build an interface for generating τ RTL trees, we can use this technique to call functions produced by the Toolkit to generate binary instructions from τ RTLs. This is possible due to the declarative nature of both the τ RTL grammar and the matching SLED aspects. However, the aspects themselves do not provide sufficient information to generate the necessary code. Instead, the processor that takes the aspects and generates the interface must include whatever information is necessary to generate the C functions from the mappings of C signatures to τ RTLs. This is appropriate though since this information is application dependent.

5.4 Summary

In this chapter, we have presented a new framework for developing descriptions of computing systems. The CSDL system facilitates the construction of descriptions that can be shared among many software applications. As a goal, we would like to build application-independent descriptions. In practice, this is not always feasible. CSDL recognizes this fact and provides the appropriate mechanisms to software developers.

Because it is difficult, if even possible, 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.

CHAPTER 6

CONCLUSIONS

This dissertation presents a framework for building application-independent descriptions of computing systems for use in retargetable software. We illustrate the framework by developing the core description component (τ RTL), an optional calling convention component (CCL), and the mechanism we use to extend extant descriptions (CSDL). In addition, another CSDL language, called PLUNGE, is being developed for describing the pipeline structure of a machine [Mil99]. The features of the Computing System Description Language directly support the evolution that machine descriptions experience.

Description systems have traditionally been tailored for a single application to make retargeting of the application more manageable. In this spirit, machine descriptions generally forgo application independence in trade for ease of application implementation. Frequently, this results in machine descriptions that actually describe the process that the application is performing (code generation, optimization, binary decoding, *etc.*) rather than solely the characteristics of the target machine. CSDL recognizes that while machine descriptions usually have information that is valuable to multiple applications, they also commonly contain details that are of use to only one application. Rather than forbidding such application dependencies, CSDL provides mechanisms to denote application-specific information.

At the center of CSDL is the core machine description. The core builds upon a proven method of modeling the effects of machine instructions: register transfer lists. RTLs are then extended to address their known shortcomings. The result is a mature, application-independent representation for concisely describing the semantics of machine instructions.

Applications that use instruction descriptions manipulate an internal or intermediate representation of instructions. CSDL provides the τ RTL representation to aid in building more robust implementations. It is the nature of these instruction “manipulators” that instructions must be added, removed, or rearranged. Often, the goal is to build semantically equivalent sequences of instructions. τ RTL provides a natural type system that makes it possible for software to easily identify malformed τ RTL expressions. Unlike other instruction representations, τ RTL uses a type system better suited to the kinds of objects being manipulated. Types are not associated with objects, as they are in high-level programming languages, but instead with operators. This reflects the process that is used when translating instructions from a high-level language to a low-level language. The types of objects in high-level languages are used to determine the types of the operations to use in implementing the high-level language expressions in the low-level form. This abstraction is appropriate for high-level languages, but not for software that manipulates a representation of low-level machine-language instructions.

When designing an instruction representation, one must always balance between a particular application’s need for detailed information and the appropriate level of abstraction to present to all applications. If too much detail is chosen, an operation’s effect cannot be easily determined by the software that manipulates the instructions. If too little detail is chosen, the application is starved for information. In the CSDL core, we balance these competing demands by providing just the right level of detail in τ RTL for many applications, while providing access to additional detail through the μ RTL definition of operations. This acknowledges the fact that many applications, such as optimizers, require some information about the instructions they manipulate, while other applications, such as simulators, need more detailed information about the operation of an instruction. This approach is hierarchical, and, although this is a common approach in computer hardware description languages, it has not been applied to the machine description domain.

Other components beyond the core can be added to CSDL descriptions. A component unique to CSDL is the module that specifies procedure calling conventions. Unlike many other target machine characteristics, the implementation of procedure call combines language dependent-information with machine dependent-information. As such, the part of an application’s implementation that works with the calling convention is among the most

difficult to retarget. Generally, the cause is that conventions are inappropriately modeled as sequences of target machine instructions. An important contribution of this thesis is the identification of the procedure calling convention and the procedure calling sequence as separate concepts that can—and should—be separated in an application’s implementation.

Once a procedure calling convention is established, software must be built that adheres to the convention. Although traditional programming techniques are generally used, programming languages do not provide the appropriate abstractions for building robust implementations of calling conventions. We demonstrate this by identifying errors in the implementations of mature production compilers. CCL provides the appropriate abstractions to accurately and succinctly specify the agreement of how information must be transmitted between caller and callee. By using CCL, we have developed convention implementations like no other: robust, bug-free, with provable properties.

Using formal specifications can often simplify implementations as well. Our experience with CCL specifications reinforces this observation. Before CCL’s integration with our compiler, the compiler’s hand-written calling-sequence generator was error prone and usually required examination several times during the process of retargeting the compiler. The result of CCL’s integration is a small, simple implementation of an FSA whose actions are directed by a generated table. Because the purpose of the implementation is formally defined, the code is easy to write and debug. Further, since the machine-specific details are encoded in the table, the result is machine-independent code that need not be revisited at all when the compiler is retargeted.

CCL’s formal specifications enable us to leverage off the language’s theoretical foundation in many ways. For example, these descriptions are truly multipurpose; they can be used to generate calling sequence code generators or to build compiler test suite generators. These two applications of CCL exercise the specifications in fundamentally different ways. This is only possible because CCL achieves an unusually high degree of application-independence due to the separation of convention from sequence. The result is that from CCL descriptions we can automatically generate test suites that are target-machine sensitive. Because target machine information is considered when constructing the test suite, the test suite generator can isolate many potential weaknesses of a compiler’s convention implementation. Finally,

perhaps the most exciting benefit is that compiler writers need not rework this aspect of a compiler's test suite when the compiler is retargeted.

Since CCL uses FSA's to model the transmission of information between caller and callee, the problem of generating a test suite for a compiler's convention generator can be reduced to exercising the FSA that is implicit (or explicit in our case) in a compiler's implementation. However, exercising a machine that recognizes strings from an infinite language poses a serious problem: not all inputs can be tested. This is a well known problem that has been investigated in many domains. We present a new solution to the infinite test problem. By using *transition-pairing*, we can select inputs that consider the context of a FSA transition's use. The result is a drastic reduction in the input test language's size, while maintaining a high degree of test coverage confidence. This method of test vector selection automatically identifies boundary conditions that are the most likely to pose problems in hand-written compiler implementations.

Generating exhaustive test suites, as we have done with CCL, is sufficient to identify errors and provide examples that will reproduce the errors. This only solves half the problem. In order to correct the implementation, one must isolate and diagnose the cause of the error. The test suite generator is capable of automatic diagnosis of some types of common errors. The diagnosis suggests the type of error in calling convention terms. When automatic diagnosis cannot be provided, usually the set of tests that fail can suggest to the compiler writer the nature of the implementation error. Best of all, the test suite and the diagnostic driver can be used in environments that do not to use CCL generated implementations, or even in cases where a CCL generator is not available to generate the test suite.

Both the τ RTL and CCL components are bound together by CSDL. CSDL provides an extremely extensible environment for collecting target machine information. As description systems mature, CSDL can accommodate descriptions of new features through its description components. Applications that view the same abstractions in different ways can include these alternative views in CSDL components without impacting existing applications or descriptions. The result is a flexible environment that gives applications room to grow and evolve.

In summary, we have presented a system for building computing system descriptions that can be used by more than a single application. τ RTL descriptions further the state-of-the-

art in specifying effects of machine instructions. μ RTL provides a hierarchical solution for applications that need more detailed information about an instruction's effect. The CCL specification language is the first work to formally describe procedure calling conventions. It also represents one of the first languages to be used by more than a single application. The enclosing CSDL environment houses the first description system to recognize that specifications must evolve and that specifications will frequently include application-dependent features. As these specifications grow, CSDL is poised to grow with them.

APPENDIX A

CSDL DESCRIPTIONS

This appendix contains the CSDL core descriptions for the MIPS and Motorola M68020.

A.1 The MIPS Core Description

```
inst → load
      | store
      | arithmetic
      | mult
      | branch
      | float
      | special
      |  $reg' \leftarrow reg''_{bs}$ ;
      |  $dreg' \leftarrow dreg''_{b,64}$ ;
      |  $reg \leftarrow expr_w$ ;
      |  $reg' \leftarrow \underline{abs}(reg''_{s,32})_w$ ;
      |  $reg' \leftarrow -(reg''_{s,32})_{s,32}$ ;
      |  $reg' \leftarrow -(reg''_w)_w$ ;
      |  $reg' \leftarrow \neg(reg''_{bs})_{bs}$ ;
      |  $dreg'_d \leftarrow dreg''_d \text{ aop } dreg'''_d$ ;
      |  $reg \leftarrow bsimm32$ ;
      |  $reg \leftarrow 0_{bs}$ ;
      |  $reg'_{bs} \leftarrow reg''_{bs} \wedge \Delta(imm16)_{bs}$ ;
      //
w → u,32 //
bs → b,32 //
d → f,64 //
FC → fc //
PC → pc //
HI → hi //
LO → lo //
shamt → constant
      //
imm →  $\Delta(\text{constant}_{s,16})$ 
      | constant
```

```

|   local
|   label
|   locals,32 + constants,32
|   labels,32 + constants,32
//
expr → Δ(constants,16)
|   constant
//
offset → Δ(constants,16)
|   constant
//
reloc → global
|   local
|   label
//
addr → regw
|   (regw + offsetw)w
|   relocw
|   (regw + relocw)w
|   (relocw + offsetw)w
|   (regw + (relocw + offsetw)w)w
//
imm16 → constants,16
|   locals,16
|   (locals,16 + constants,16)s,16
//
bsimm16 → constantb,16
|   labelb,16
//
bsimm32 → globalb,32
|   labelb,32
//
imm32 → globalb,32
|   (globalw + Δ(constants,16)w)w
//
bimm16 → constants,16
//
imm26 → constant
|   global
//
regimm → reg
//
regno → constantu,5 //
dregno → constantu,5 //
freg → f[dregno]31..0 //
dreg → f[dregno] //
reg → r[regno] //
reg0 → reg
|   0
//
mem → m[addr]

```

```

//
immlop → ^
|      v
|      ⊕
//
lop    → immlop
|      nor
//
frelop → =
|      ≤
|      <
//
relop  → =
|      ≠
//
signedrelop → ≤
|      <
|      >
|      ≥
//
shiftop → ⇐
|      ⇒
//
immaop → +
|      -
//
aop    → +
|      -
|      ×
|      ÷
//

load   → reg ← membs;
|      reg ← Δ(mems,8)s,32;
|      reg ← Δ(mems,16)s,32;
|      reg ← addr;
|      reg ← Δ(memu,8)w;
|      reg ← Δ(memu,16)w;
|      dreg ← memb,64;
|      freg ← memb,32;
|      reg ← imm32;
//

store  → mem ← regbs;
|      mem ← 0w;
|      mem ← 0u,16;
|      mem ← 0u,8;
|      mem ← Δ(regb,32)b,8;
|      mem ← Δ(regb,32)b,16;
|      mem ← dregb,64;
|      mem ← fregb,32;
//

arithmetic → reg's,32 ← reg''s,32 immaop imms,32;
|      reg'w ← reg''w immaop immw;

```

```

|    $reg' \leftarrow imm_{s,32};$ 
|    $reg' \leftarrow imm_w;$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ immlop } imm_{bs};$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \text{ aop } reg'''_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \text{ aop } reg'''_w;$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ lop } reg'''_{bs};$ 
|    $reg' \leftarrow \neg((reg''_{bs} \vee reg'''_{bs})_{bs})_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{s,32} < reg'''_{s,32};$ 
|    $reg'_{bs} \leftarrow reg''_w < reg'''_w;$ 
|    $reg'_{bs} \leftarrow reg''_{s,32} \text{ signedrelop } constant_{s,32};$ 
|    $reg'_{bs} \leftarrow reg''_{s,32} \text{ signedrelop } reg'''_{s,32};$ 
|    $reg'_{bs} \leftarrow reg''_w \text{ signedrelop } constant_w;$ 
|    $reg'_{bs} \leftarrow reg''_w \text{ signedrelop } reg'''_w;$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ relop } constant_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ relop } reg'''_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ rol } constant_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ rol } reg'''_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ ror } constant_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ ror } reg'''_{bs};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ shiftopt } sham_{u,5};$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \Rightarrow sham_{u,5};$ 
|    $reg'_{bs} \leftarrow reg''_{bs} \text{ shiftopt } \Delta(reg'''_w)_{u,5};$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \Rightarrow \Delta(reg'''_w)_{u,5};$ 
//
mult →  $HI \leftarrow \Delta((reg'_{s,32} \times reg''_{s,32})_{s,64})_{s,32}; LO \leftarrow \Delta((reg'_{s,32} \times reg''_{s,32})_{s,64})_{s,32};$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \times reg'''_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \times reg'''_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \times constant_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \times constant_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \text{ mulo } reg'''_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \text{ mulou } reg'''_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \text{ mulo } constant_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \text{ mulou } constant_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \div reg'''_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \div reg'''_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \div constant_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \div constant_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \% reg'''_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \% reg'''_w;$ 
|    $reg'_{s,32} \leftarrow reg''_{s,32} \% constant_{s,32};$ 
|    $reg'_w \leftarrow reg''_w \% constant_w;$ 
|    $reg \leftarrow HI_{bs};$ 
|    $reg \leftarrow LO_{bs};$ 
|    $LO \leftarrow reg_{bs};$ 
|    $HI \leftarrow reg_{bs};$ 
//
jtarget →  $\Delta(constant_{b,26})_{bs}$ 
|    $reg_{bs}$ 
|   labelbs
|   globalbs
//

```

```

btarget →  $\Delta((PC_w + \Delta(bimm16)_w)_w)_{bs}$ 
| labelbs
//
balrelop → <
| ≥
//
branch → PC ← jtarget,
| PC ← jtarget, r[31u,5] ← PCbs;
| PC ← regbs;
| PC ← regbs; r[31u,5] ← PCbs;
| PC ←  $\zeta((reg'_{bs} \text{ relop } reg''_{bs})_{bs}, btarget, PC_{bs})_{bs}$ ;
| PC ←  $\zeta((reg_{s,32} \text{ signedrelop } regimm_{s,32})_{bs}, btarget, PC_{bs})_{bs}$ ;
| PC ←  $\zeta((reg_w \text{ signedrelop } regimm_w)_{bs}, btarget, PC_{bs})_{bs}$ ;
| PC ←  $\zeta((reg_{bs} \text{ relop } 0_{bs})_{bs}, btarget, PC_{bs})_{bs}$ ;
| PC ←  $\zeta((reg_{0s,32} \text{ signedrelop } 0_{s,32})_{bs}, btarget, PC_{bs})_{bs}$ ;
| PC ←  $\zeta((reg_{s,32} \text{ balrelop } 0_{s,32})_{bs}, btarget, PC_{bs})_{bs}; r[31_{u,5}] \leftarrow PC_{bs}$ ;
| PC ←  $\zeta((FC_{s,32} = 0_{s,32})_{bs}, btarget, PC_{bs})_{bs}$ ;
| PC ←  $\zeta((FC_{s,32} = 1_{s,32})_{bs}, btarget, PC_{bs})_{bs}$ ;
//
special → break(constantb,20)
| tlbp
| tlbr
| tlbwr
| tlbwl
| rfe
| syscall
//
float → dreg ← (r[regno]bs:r[regno']bs)b,64; { ((regno' % 2) = 0) ∧ (regno' = (regno" + 1)) }
| (r[regno]:r[regno']) ← dregb,64;
| f[dregno]63..32 ← reg'bs; f[dregno]31..0 ← reg''bs; { ((regno' % 2) = 0) ∧ (regno' = (regno" + 1)) }
| reg'bs ← f[dregno]63..32; reg''bs ← f[dregno]31..0; { ((regno' % 2) = 0) ∧ (regno' = (regno" + 1)) }
| freg ← regbs;
| reg ← fregbs;
| fregbs ← mem;
| dregb,64 ← mem;
| membs ← freg;
| memb,64 ← dreg;
| dregd ←  $\Delta(freg_{f,32})$ ;
| fregf,32 ←  $\Delta(dreg_d)$ ;
| freg'f,32 ←  $\Delta(freg''_{s,32})$ ;
| fregf,32 ←  $\Delta(reg_{s,32})$ ;
| dregf,64 ←  $\Delta(freg_{s,32})$ ;
| dregf,64 ←  $\Delta(reg_{s,32})$ ;
| freg's,32 ←  $\Delta(freg''_{f,32})$ ;
| fregs,32 ←  $\Delta(dreg_{f,64})$ ;
| dreg'd ← dreg''d aop dreg'''d;
| freg'f,32 ← freg''f,32 aop freg'''f,32;
| freg'f,32 ←  $-(freg''_{f,32})$ ;
| dreg'f,32 ←  $-(dreg''_{f,32})$ ;
| dreg' ← dreg''b,64;
| freg' ← freg''b,32;

```



```

|  $freg'_{f,32} \leftarrow \underline{abs}(freg''_{f,32});$ 
|  $dreg'_{f,32} \leftarrow \underline{abs}(dreg''_{f,32});$ 
|  $FC_{bs} \leftarrow dreg'_d \text{ frelop } dreg''_d;$ 
|  $FC_{bs} \leftarrow freg'_{f,32} \text{ frelop } freg''_{f,32};$ 
|  $dreg \leftarrow \mathbf{fconstant}_d;$ 
|  $freg \leftarrow \mathbf{fconstant}_{f,32};$ 
//

```

A.2 The Motorola M68020 Core Description

```

inst → move
| arith
| logical
| shift
| bitfield
| branch
//
PC → pc //
CC → cc //
SP → a[7] //
regno → constantu,3 //
dreg → d[regno] //
areg → a[regno] //
pcareg → areg
| PC //
reg → areg
| dreg
//
sz → 8
| 16
| 32
//
bbwl → b,8
| b,16
| b,32
//
bwl → u,8
| u,16
| u,32
//
wl → u,16
| u,32
//
L → u,32 //
sbwl → s,8
| s,16
| s,32
//
scale → 1
| 2

```

```

| 4
| 8
//
swl → s,16
| s,32
//
index →  $(\Delta(\text{reg}_{sbwl})_L \times \Delta(\text{scale}_{u,8})_L)_L$ 
|  $\Delta(\text{reg}_{sbwl})_L$ 
//
windex →  $(\Delta(\text{reg}_{swl})_L \times \Delta(\text{scale}_{u,8})_L)_L$ 
|  $\Delta(\text{reg}_{swl})_L$ 
//
indirect →  $(\text{areg}_L + \text{wlconst}_L)_L$ 
|  $\text{wlconst}_L$ 
|  $\text{areg}_L$ 
//
wlconst →  $\Delta(\text{constant}_{s,16})$ 
|  $\Delta(\text{constant}_{s,32})$ 
//
ea → dreg
| areg
|  $m[\text{areg}_L]$ 
|  $m[(\text{pcareg}_L + \Delta(\text{constant}_{s,16})_L)_L]$ 
|  $m[(\text{pcareg}_L + \Delta(\text{constant}_{s,8})_L)_L + \text{windex}]_L$ 
|  $m[(\text{pcareg}_L + \text{wlconst}_L)_L + \text{index}]_L$ 
|  $m[(\text{pcareg}_L + \text{index})_L]$ 
|  $m[(\text{wlconst}_L + \text{index})_L]$ 
|  $m[(\text{pcareg}_L + \text{wlconst}_L)_L]$ 
|  $m[\text{index}]$ 
|  $m[\text{pcareg}_L]$ 
|  $m[\text{wlconst}_L]$ 
|  $m[(\text{areg}'_L \leftarrow \text{areg}'_L + \text{scale}_L) - \text{scale}_L]_L$ 
|  $m[\text{areg}'_L \leftarrow \text{areg}'_L - \text{scale}_L]$ 
|  $m[0_L]$ 
|  $m[(m[\text{indirect}]_L + \text{index})_L + \text{wlconst}_L]_L$ 
|  $m[(m[\text{indirect}]_L + \text{wlconst}'_L)_L]$ 
|  $m[(m[\text{indirect}]_L + \text{index})_L]$ 
|  $m[m[\text{indirect}]_L]$ 
|  $m[\text{indirect}]$ 
|  $m[\text{wlconst}_L]$ 
|  $m[\text{index}]$ 
|  $m[(m[(\text{indirect} + \text{index})_L]_L + \text{wlconst}_L)_L]$ 
|  $m[\Delta(\text{constant}_{s,16})_L]$ 
|  $m[\text{constant}_L]$ 
| constant
//
imm → constant
//
wl → 16
| 32
//
bw → 8

```

```

| 16
//
n → s
| u
//
move →
|  $dreg'_{b,32} \leftarrow dreg''; dreg'_{b,32} \leftarrow dreg'$ ;
|  $dreg_{b,32} \leftarrow areg; areg_{b,32} \leftarrow dreg$ ;
|  $areg'_{b,32} \leftarrow areg''; areg'_{b,32} \leftarrow areg'$ ;
|  $areg_{b,32} \leftarrow ea$ ;
|  $m[(SP_L - 4_L)_L]_{b,32} \leftarrow areg'; areg'_L \leftarrow SP_L - 4_L$ ;
|  $SP_L \leftarrow (SP_L - 4_L)_L + \Delta(\mathbf{constant}_{swl})_L$ ;

|  $ea'_{bbwl} \leftarrow ea''; setcc$ ;
|  $areg'_{b,32} \leftarrow m[areg'_L]; SP_L \leftarrow areg'_L + 4_L$ ;
|  $m[(SP_L - 4_L)_L]_{b,32} \leftarrow ea; SP_L \leftarrow SP_L - 4_L$ 
//
setcc →  $CC_{b,5} \leftarrow //$ 
aop →
| +
| -
//
arith →
|  $dreg'_{bwl} \leftarrow ea_{bwl} aop dreg'_{bwl}; setcc$ ;
|  $ea'_{bwl} \leftarrow dreg_{bwl} aop ea'_{bwl}; setcc$ ;
|  $areg'_{wl} \leftarrow ea_{wl} aop areg'_{wl}$ ;
|  $dreg'_{bwl} \leftarrow \text{addx}(dreg''_{bwl}, dreg'_{bwl}); setcc$ ;
|  $ea'_{bwl} \leftarrow 0; setcc$ ;
|  $cc_{b,5} \leftarrow dreg_{sbwl} - ea_{sbwl}$ ;
|  $cc_{b,5} \leftarrow areg_{swl} - ea_{swl}$ ;
|  $dreg'^{15..0}_{n,16} \leftarrow dreg'_{n,16} \div ea'_{n,16}$ ;
|  $dreg'^{31..16}_{n,16} \leftarrow dreg'_{n,16} \bmod ea'_{n,16}; setcc$ ;
|  $dreg'_{n,32} \leftarrow dreg'_{n,32} \div ea'_{n,32}; setcc$ ;
|  $dreg'_{n,32} \leftarrow (dreg'':dreg')_{n,64} \div ea'_{n,32}$ ;
|  $dreg''_{n,32} \leftarrow (dreg'':dreg')_{n,64} \bmod ea'_{n,32}; setcc$ ;
|  $dreg'_{n,32} \leftarrow dreg'_{n,32} \div ea'_{n,32}$ ;
|  $dreg''_{n,32} \leftarrow dreg'_{n,32} \bmod ea'_{n,32}; setcc$ ;

|  $dreg'_{s,32} \leftarrow \Delta(dreg'_{s,bwl})$ ;
|  $dreg'_{s,16} \leftarrow \Delta(dreg'_{s,8})$ ;
|  $dreg'_{n,wl} \leftarrow dreg'_{n,wl} \times ea'_{n,wl}; setcc$ ;
|  $(dreg'':dreg'')_{n,64} \leftarrow ea_{n,32} \times dreg''_{n,32}; setcc$ ;
|  $ea'_{sbwl} \leftarrow -ea'_{sbwl}; setcc$ ;
|  $ea'_{sbwl} \leftarrow \text{negx}(ea'_{sbwl}); setcc$ ;
//
logop → ^
| ⊕
| ∨
//
logical →  $dreg'_{bbwl} \leftarrow ea_{bbwl} logop dreg'_{bbwl}$ 
|  $ea'_{bbwl} \leftarrow dreg_{bbwl} logop ea'_{bbwl}$ 

```

```

|    $ea'_{bbwf} \leftarrow \neg(ea'_{bbwf})$ 
//
ashiftop → ←
|   ⇒
//
shifto → ←
|   ⇒
|   rol
|   ror
|   roxl
|   roxr
//
al     → b
|     s
//
shift  →
|    $dreg'_{sbwf} \leftarrow dreg'_{sbwf} \text{ ashiftop } dreg'_{u,5};$ 
|    $dreg'_{sbwf} \leftarrow dreg'_{sbwf} \text{ ashiftop } imm_{u,5};$ 
|    $ea'_{s,16} \leftarrow ea'_{s,16} \text{ ashiftop } 1_{u,5};$ 
|    $dreg'_{bbwf} \leftarrow dreg'_{bbwf} \text{ shifto } dreg'_{u,5};$ 
|    $dreg'_{bbwf} \leftarrow dreg'_{bbwf} \text{ shifto } imm_{u,5};$ 
|    $ea'_{b,16} \leftarrow ea'_{b,16} \text{ shifto } 1_{u,5};$ 
|    $dreg'^{15..0}_{b,16} \leftarrow dreg'^{31..16}; dreg'^{31..16}_{b,16} \leftarrow dreg'^{15..0};$ 
//
bfop   → bfexts
|     bfextu
|     bfins
//
bitfield →  $dreg'_{b,32} \leftarrow bfop(ea_{b,32}, \text{constant}'_{u,5}, \text{constant}''_{u,5}); setcc;$ 
//
relop  → =
|     ≠
|     ≤
|     <
|     >
|     ≥
//
cmp    →  $CC_{u,5} \text{ relop } 0_{u,32} //$ 
branch →  $PC_L \leftarrow PC_L + \Delta(\text{label}_{sbwf})_L;$  label
|    $PC_L \leftarrow PC_L + \Delta(\text{label}_{sbwf})_L; m[SP_L]_{b,32} \leftarrow PC; SP_L \leftarrow SP_L - 4_L;$  label
|    $PC_{b,32} \leftarrow ea;$ 
|    $PC_{b,32} \leftarrow ea; m[SP_L]_{b,32} \leftarrow PC; SP_L \leftarrow SP_L - 4_L;$ 
|    $PC_L \leftarrow PC_L + 2_L;$ 
|    $PC_{b,32} \leftarrow ?(cmp_{b,1}, \text{label}_{b,32}, PC_{b,32});$  label
|    $PC_{b,32} \leftarrow ?(DB(cmp_{b,1})_{b,1}, \text{label}_{b,32}, PC_{b,32});$  label
|    $ea_{b,32} \leftarrow cmp;$  label
|    $PC_{b,32} \leftarrow m[SP_L]; SP_L \leftarrow (SP_L + 4_L)_L + \Delta(\text{constant}_{s,16})_L;$ 
|    $PC_{b,32} \leftarrow m[SP_L]; SP_L \leftarrow SP_L + 4_L;$ 
//

```

APPENDIX B

CCL DESCRIPTIONS

This appendix contains the CCL calling convention descriptions.

2.1 The MIPS R3000 CCL Description

1. **external** SPILL_SIZE, LOCALS_SIZE
 2. **persistent** $\{r^1, r^{16:23}, r^{26:31}\}$
 3. **alias** REG_ARGS $\equiv 16$
 4. **alias** sp $\equiv r^{29}$
 5. **caller prologue**
 6. **view change**
 7. $\forall offset \in \{-\infty:\infty\}$
 8. M[sp + offset] becomes M[sp + offset + \lceil ARG_BLOCK_SIZE $\rceil^8]$
 9. **end view change**
 10. **data transfer (asymmetric)**
 11. **alias** rindex $\equiv 4:7$
 12. **alias** fpindex $\equiv 12,14$
 13. **alias** mstart \equiv sp + REG_ARGS
 14. **alias** mindex \equiv mstart: ∞
 15. **resources** $\{\langle r^{rindex}, M^{mindex} \rangle, \langle f^{fpindex}, M^{mindex} \rangle, \langle M^{mindex} \rangle\}$
 16. $\forall mem \in \{M[sp(REG_ARGS)]\}$ **set** mem.assigned \leftarrow true
 17. **internal** ARG_BLOCK_SIZE $\leftarrow \sum(\langle M[addr].size \mid addr \in \langle mindex \rangle$
 18. $\wedge M[addr].assigned \rangle)$
 19. **class** intregs $\leftarrow \langle \langle r^x \rangle \mid x \in \langle rindex \rangle \rangle$
 20. **class** intfpregs $\leftarrow \langle \langle r^x \rangle \mid x \in \langle rindex \rangle \wedge x \bmod 2 = 0 \rangle$
 21. **class** fpfpregs $\leftarrow \langle \langle f^x \rangle \mid x \in \langle fpindex \rangle \rangle$
 22. **class** mem $\leftarrow \langle \langle M^{loc} \rangle \mid loc \in \langle mindex \rangle \wedge loc \bmod 4 = 0 \rangle$
 23. **class** aligned_mem $\leftarrow \langle \langle M^{loc} \rangle \mid loc \in \langle mindex \rangle \wedge loc \bmod 8 = 0 \rangle$
 24. **class** struct_mem $\leftarrow \langle \langle r^x, M^{mstart} \rangle \mid x \in \langle rindex \rangle \rangle$
 25. **class** aligned_struct_mem $\leftarrow \langle \langle r^x, M^{mstart} \rangle \mid x \in \langle rindex \rangle \wedge x \bmod 2 = 0 \rangle$
 26. $\exists reg \in \{reg \mid reg \in \{f^{12}\} \wedge reg.assigned\} \Rightarrow$ **set** r^4 .unavailable \leftarrow true
 27. $\exists reg \in \{reg \mid reg \in \{f^{12}\} \wedge reg.assigned\} \Rightarrow$ **set** r^5 .unavailable \leftarrow true
 28. $\exists reg \in \{reg \mid reg \in \{f^{14}\} \wedge reg.assigned\} \Rightarrow$ **set** r^6 .unavailable \leftarrow true
 29. $\exists reg \in \{reg \mid reg \in \{f^{14}\} \wedge reg.assigned\} \Rightarrow$ **set** r^7 .unavailable \leftarrow true
 30. $\exists reg \in \{reg \mid reg \in \{r^{4:5}\} \wedge reg.assigned\} \Rightarrow$ **set** f^{12} .unavailable \leftarrow true
-

```

31.       $\exists reg \in \{reg \mid reg \in \{r^{6:7}\} \wedge reg.assigned\} \Rightarrow set f^{14}.unavailable \leftarrow true$ 
32.       $\forall argument \in \langle ARG^{1:ARG\_TOTAL} \rangle$ 
33.          map argument  $\rightarrow argument.type \perp \{$ 
34.              byte, word, longword:  $\langle intregs, mem \rangle,$ 
35.              struct:      argument.size  $\perp \{$ 
36.                  1,2,3,4,5,6,7:  $\langle struct\_mem, mem \rangle,$ 
37.                  default:       $\langle aligned\_struct\_mem, aligned\_mem \rangle$ 
38.               $\},$ 
39.              float, double:  $ARG^1.type \perp \{$ 
40.                  struct, byte, word, longword:  $\langle intfpregs, aligned\_mem \rangle,$ 
41.                  float, double:       $\langle fpfpregs, aligned\_mem \rangle$ 
42.               $\}$ 
43.           $\}$ 
44.      end data transfer
45.  end caller prologue
46.  callee prologue
47.      view change
48.           $\forall offset \in \{-\infty; \infty\}$ 
49.               $M[sp + offset] \text{ becomes } M[sp + offset + \lceil SPILL\_SIZE +$ 
50.                   $LOCALS\_SIZE + NVSIZE \rceil^8]$ 
51.      end view change
52.  end callee prologue
53.  callee epilogue
54.      data transfer (asymmetric)
55.      resources  $\{ \langle r^2 \rangle, \langle f^0 \rangle \}$ 
56.       $\exists return \in \langle RVAL^{1:RVAL\_TOTAL} \rangle \Rightarrow$ 
57.          map return  $\rightarrow return.type \perp \{$ 
58.              byte, word, longword:  $\langle \langle \langle r^2 \rangle \rangle \rangle,$ 
59.              float, double:       $\langle \langle \langle f^0 \rangle \rangle \rangle,$ 
60.              struct:               $\uparrow(\langle \langle \langle r^2 \rangle \rangle \rangle)$ 
61.           $\}$ 
62.      end data transfer
63.  end callee epilogue

```

2.2 The M68020 CCL Description

Description for the M68020

1. **external** SPILL_SIZE, LOCALS_SIZE
 2. **persistent** { $d^{2:7}, a^{2:7}$ }
 3. **alias** sp $\equiv a^7$
 4. **caller prologue**
 5. **view change**
 6. $\forall offset \in \{-\infty; \infty\}$
 7. M[sp + offset] becomes M[sp + offset + $\lceil ARG_BLOCK_SIZE \rceil^4$]
 8. **end view change**
 9. **data transfer (asymmetric)**
-
- All arguments, regardless of type, are saved on the stack, just below the local variable space. The extra 4 bytes is included because the call instruction pushes the return address on the stack, causing an additional 4 bytes to be calculated.*
-
10. **alias** mindex $\equiv sp:\infty$
 11. **resources** { $\langle M^{mindex} \rangle$ }
 12. **class** mem $\leftarrow \langle \langle M^{loc} \rangle \mid loc \in \langle mindex \rangle \rangle$
 13. **internal** ARG_BLOCK_SIZE $\leftarrow \sum(\langle M[addr].size \mid addr \in \langle mindex \rangle \wedge$
 14. M[addr].assigned)
 15. $\forall argument \in \langle ARG^{1:ARG_TOTAL} \rangle$
 16. **map** argument $\rightarrow \langle mem \rangle$
 17. **end data transfer**
 18. **end caller prologue**
 19. **callee prologue**
 20. **view change**
 21. $\forall offset \in \{-\infty; \infty\}$
 22. M[sp + offset] becomes M[sp + offset + 4 + $\lceil SPILL_SIZE + LOCALS_SIZE +$
 23. NVSIZ^E \rceil^4]
 24. **end view change**
 25. **end callee prologue**
 26. **callee epilogue**
 27. **data transfer (asymmetric)**
 28. **resources** { $\langle d^{0:1} \rangle$ }
 29. $\exists return \in \langle RVAL^{1:RVAL_TOTAL} \rangle \Rightarrow$
 30. **map** return $\rightarrow return.type \perp \{$
 31. byte, word, longword, float, double: $\langle \langle \langle d^0 \rangle \rangle \rangle$,
 32. struct: $\uparrow(\langle \langle \langle d^0 \rangle \rangle \rangle)$
 33. }
 34. **end data transfer**
 35. **end callee epilogue**
-

2.3 The M88100 CCL Description

Description for the M88100

1. **external** ARG_SIZE, SPILL_SIZE, LOCALS_SIZE
 2. **persistent** $\{r^1, r^{14:31}\}$
 3. **alias** $sp \equiv r^{31}$
 4. **alias** $mem_bytes \equiv 32$
 5. **caller prologue**
 6. **data transfer (asymmetric)**
 7. **alias** $rindex \equiv 2:9$
 8. **alias** $mstart \equiv sp + mem_bytes$
 9. **alias** $mindex \equiv mstart:\infty$
 10. **resources** $\{\langle r^{rindex}, M^{mindex} \rangle\}$
 11. **class** $regs \leftarrow \langle \langle r^x \rangle \mid x \in \langle rindex \rangle \rangle$
 12. **class** $mem \leftarrow \langle \langle M^{loc} \rangle \mid loc \in \langle mindex \rangle \wedge loc \bmod 4 = 0 \rangle$
 13. **class** $aligned_mem \leftarrow \langle \langle M^{loc} \rangle \mid loc \in \langle mindex \rangle \wedge loc \bmod 8 = 0 \rangle$
 14. **internal** ARG_BLOCK_SIZE $\leftarrow \sum(\langle M[addr].size \mid addr \in \langle mindex \rangle \wedge$
15. $M[addr].assigned \rangle)$
-
- This explicitly requires that an additional 32 bytes be allocated on the stack if there is any stack space allocated by the caller.*
-
16. $\exists memarg \in \{memory \mid memory \in \langle M[mstart] \rangle \wedge memory.assigned\} \Rightarrow$
 17. $stackloc \in \{M[loc] \mid loc \in \{sp(mem_bytes)\}\} \text{ set } stackloc.assigned \leftarrow true$
 18. $\forall argument \in \langle ARG^{1:ARG_TOTAL} \rangle$
 19. **map** $argument \rightarrow argument.type \perp \{$
 20. $byte, word, longword: \langle regs, mem \rangle,$
 21. $float, double: \langle regs, aligned_mem \rangle,$
 22. $struct: \langle aligned_mem \rangle$
 23. $\}$
 24. **end data transfer**
 25. **end caller prologue**
 26. **callee prologue**
 27. **view change**
 28. $\forall offset \in \{-\infty:\infty\}$
 29. $M[sp + offset] \text{ becomes } M[sp + offset + \lceil SPILL_SIZE + LOCALS_SIZE +$
 30. $NVSIZE + ARG_BLOCK_SIZE \rceil 8]$
 31. **end view change**
 32. **end callee prologue**
 33. **callee epilogue**
 34. **data transfer (asymmetric)**
 35. **resources** $\{\langle r^{2:3} \rangle\}$
 36. $\exists return \in \langle RVAL^{1:RVAL_TOTAL} \rangle \Rightarrow$
 37. **map** $return \rightarrow return.type \perp \{$
 38. $byte, word, longword, float, double: \langle \langle \langle r^2 \rangle \rangle \rangle,$
 39. $struct: \uparrow(\langle \langle \langle r^2 \rangle \rangle \rangle)$
 40. $\}$
 41. **end data transfer**
 42. **end callee epilogue**
-

2.4 The DEC VAX-11 CCL Description

Description for the DEC VAX-11

1. external SPILL_SIZE, LOCALS_SIZE
2. persistent { $r^{6:15}$ }
3. alias sp $\equiv r^{14}$
4. caller prologue
 5. view change
 6. $\forall offset \in \{-\infty:\infty\}$ M[sp + offset] becomes M[sp + offset + \lceil ARG_BLOCK_SIZE $\rceil^8]$
 7. end view change
 8. data transfer (asymmetric)
 9. alias mindex \equiv sp+4: ∞
 10. resources { $\langle M^{\text{mindex}} \rangle$ }
 11. class mem $\leftarrow \langle \langle M^{\text{loc}} \mid loc \in \langle \text{mindex} \rangle \rangle$
 12. internal ARG_BLOCK_SIZE $\leftarrow \sum(\langle M[\text{addr}].\text{size} \mid \text{addr} \in \langle \text{mindex} \rangle \wedge$
 M[addr].assigned)
 13. $\forall argument \in \langle ARG^{1:ARG_TOTAL} \rangle$
 14. map argument \rightarrow argument.type \perp {
 15. byte, word, longword, float, double: $\langle \text{mem} \rangle$,
 16. struct: $\uparrow(\langle \text{mem} \rangle)$
 17. }
 18. end data transfer
19. end caller prologue
20. callee prologue
 21. view change
 22. $\forall offset \in \{-\infty:\infty\}$
 23. M[sp + offset] becomes M[sp + offset + \lceil SPILL_SIZE + LOCALS_SIZE +
 NVSIZE $\rceil^8]$
 24. end view change
25. end callee prologue
26. callee epilogue
 27. data transfer (asymmetric)
 28. resources { $\langle r^{0:1} \rangle$ }
 29. $\exists return \in \langle RVAL^{1:RVAL_TOTAL} \rangle \Rightarrow$
 30. map return \rightarrow return.type \perp {
 31. byte, word, longword, float, double: $\langle \langle r^0 \rangle \rangle$,
 32. struct: $\uparrow(\langle \langle r^0 \rangle \rangle)$
 33. }
 34. end data transfer
35. end callee epilogue

2.5 The SPARC CCL Description

Description for the SPARC

1. **external** SPILL_SIZE, LOCALS_SIZE
 2. **persistent** $\{r^{14:31}\}$
 3. **alias** $sp \equiv r^{14}$
 4. **internal** REG_MEM_SIZE $\leftarrow 92$
 5. **caller prologue**
 6. **data transfer (asymmetric)**
 7. **alias** $rindex \equiv 8:13$
 8. **alias** $mstart \equiv sp + REG_MEM_SIZE$
 9. **alias** $mindex \equiv mstart:\infty$
 10. **resources** $\{<r^{rindex}, M^{mindex}>\}$
 11. **class** $regsmem \leftarrow \langle\langle r^x, M^{mstart} \mid x \in <rindex>>\rangle$
 12. **class** $mem \leftarrow \langle\langle M^{loc} \mid loc \in <mindex>>\rangle$
 13. **internal** ARG_BLOCK_SIZE $\leftarrow \sum(\langle M[addr].size \mid addr \in <mindex> \wedge$
 14. $M[addr].assigned \rangle)$
 15. $\forall argument \in <ARG^{1:ARG_TOTAL}>$
 16. **map** $argument \rightarrow argument.type \perp \{$
 17. byte, word, longword, float, double: $\langle regsmem, mem \rangle,$
 18. struct: $\langle regsmem, mem \rangle$
 19. $\}$
 20. **end data transfer**
 21. **end caller prologue**
 22. **callee prologue**
 23. **view change**
 24. $\forall offset \in \{-\infty:\infty\}$
 25. $M[sp + offset]$ becomes $M[sp + offset + \lceil SPILL_SIZE + LOCALS_SIZE +$
 26. $NVSIZE + ARG_BLOCK_SIZE + 4 \rceil^8]$
 27. $\forall x \in \{8:15\}$
 28. $r[x]$ becomes $r[x + 16]$
 29. **end view change**
 30. **data transfer**
 31. **alias** $mindex \equiv sp:\infty$
 32. **resources** $\{<M^{mindex}>\}$
 33. **class** $mem \leftarrow \langle\langle M^{loc} \mid loc \in <mindex>>\rangle$
 34. **internal** NVSIZE $\leftarrow \sum(\langle M[addr].size \mid addr \in <mindex> \wedge M[addr].assigned \rangle)$
 35. $\forall register \in \langle r^x \mid x \in \langle 16:31 \rangle \wedge r^x.assigned \rangle$
 36. **map** $register \rightarrow \langle mem \rangle$
 37. **end data transfer**
 38. **end callee prologue**
 39. **callee epilogue**
 40. **data transfer (asymmetric)**
 41. **resources** $\{<r^8>, <f^0>\}$
 42. $\exists return \in <RVAL^{1:RVAL_TOTAL}> \Rightarrow$
 43. **map** $return \rightarrow return.type \perp \{$
 44. byte, word, longword: $\langle\langle\langle r^8 \rangle\rangle\rangle,$
 45. float, double: $\langle\langle\langle f^0 \rangle\rangle\rangle,$
-

```
46.          struct:      ↑(<<<r8>>>)
47.          }
48.      end data transfer
49.  end callee epilogue
```

REFERENCES

- [ADLU91] Alfred V. Aho, Anton T. Dahbura, David Lee, and M. Umit Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, November 1991.
- [Ado97a] Adobe Systems, Inc. *Adobe FrameMaker 5.5 User Guide*, August 1997.
- [Ado97b] Adobe Systems, Inc. *Frame Developer's Kit Programmer's Reference*, August 1997.
- [Ado97c] Adobe Systems, Inc. *MIF Reference*, August 1997.
- [ADR98] Andrew Appel, Jack Davidson, and Norman Ramsey. The zephyr compiler infrastructure. Distributed at Supercomputing '98, November 1998.
- [AGH+84] Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, and Eduardo Petegri-Llopart. Experience with a Graham-Glanville style code generator. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 13–24, June 1984.
- [AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.
- [AKS00] Erik R. Altman, David Kaeli, and Yaron Sheffer. Welcome to the opportunities of binary translation. *IEEE Computer*, 33(3), March 2000.
- [App96] Andrew W. Appel. Personal Communication, May 1996.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison Wesley, 1986.
-

-
- [BD88] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, July 1988.
- [BD94] Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures*, pages 105–124, March 1994.
- [BD95] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, January 1995.
- [BD96a] Mark W. Bailey and Jack W. Davidson. Reusable application-dependent machine descriptions. In *Workshop Record of The Inaugural Workshop on Compiler Support for Systems Software*, pages 77–85, February 1996.
- [BD96b] Mark W. Bailey and Jack W. Davidson. Target-sensitive construction of diagnostic programs for procedure calling sequence generators. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, May 1996. Also available as technical report CS-95-44, Department of Computer Science, University of Virginia, Charlottesville, VA 22903.
- [Ben89] Manuel E. Benitez. A global object code optimizer. Master's thesis, Department of Computer Science, University of Virginia, Charlottesville, VA, January 1989.
- [Ben94] Manuel E. Benitez. *Register Allocation and Phase Interactions in Retargetable Optimizing Compilers*. PhD thesis, Department of Computer Science, University of Virginia, May 1994.
- [BG79] P. I. P. Boulton and J. R. Goguen. A machine description language. *Computer Journal*, 22(2):132–135, May 1979.
- [BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–240, June 1991.
- [BN71] C. Gordon Bell and Allen Newell. Computer structures: Readings and examples. In *The PMS and ISP descriptive systems*, pages 15–36. McGraw-Hill, 1971.
- [Bra91] David Gordon Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors*. PhD thesis, University of Washington, 1991.
-

-
- [Cat78] Roderic G. G. Cattell. Using machine descriptions for automatic derivation of code generators. In *Proceedings of the Jerusalem Conference on Information Technology*, pages 503–507, 1978.
- [Cat80] Roderic G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, April 1980.
- [CE00] Cristina Cifuentes and Mike Van Emmerik. Uqbt: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.
- [CER99] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourceable and retargetable binary translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 280–291, Atlanta, GA, October 1999. IEEE-CS Press.
- [CH94a] Todd A. Cook and Ed Harcourt. A functional specification language for instruction set architectures. In *International Conference on Computer Languages*, pages 11–19, May 1994.
- [CH94b] Todd A. Cook and Ed Harcourt. Instruction set architecture specification. Submitted to TOCS, February 1994.
- [Coe89] David R. Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, 1989.
- [Coo94] Todd A. Cook. *Instruction Set Architecture Specification*. PhD thesis, North Carolina State University, April 1994.
- [Das89] Subrata Dasgupta. *Computer Architecture: A Modern Synthesis, Volume 2: Advanced Topics*. John Wiley and Sons, 1989.
- [Dav85] Jack W. Davidson. Simple machine description grammars. Technical Report CS-85-22, Department of Computer Science, University of Virginia, Charlottesville, VA, November 1985.
- [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.
- [DF84a] 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.
- [DF84b] Jack W. Davidson and Christopher W. Fraser. Register allocation and exhaustive peephole optimization. *Software-Practice and Experience*, 14(9):857–866, September 1984.
-

-
- [Dig78] Digital Equipment Corporation. *VAX Architecture Handbook*. Digital Equipment Corporation, 1978.
- [Dig93] Digital Equipment Corporation. *Calling Standard for AXP Systems*. Digital Equipment Corporation, Maynard, MA, July 1993.
- [DW91] Jack W. Davidson and David B. Whalley. Methods for saving and restoring register values across function calls. *Software-Practice and Experience*, 21(2):149–165, February 1991.
- [FH91] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software-Practice and Experience*, 21(9), 1991.
- [FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings, 1995.
- [FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg – fast optimal instruction selection and tree parsing. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 27(4):68–76, April 1992.
- [Fra77a] Christopher W. Fraser. A knowledge-based code generator generator. *SIGPLAN Notices*, 12(8):126–129, 1977.
- [Fra77b] Christopher Warwick Fraser. *Automatic Generation of Code Generators*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1977.
- [Fra93] Christopher W. Fraser. Personal Communication, November 1993.
- [FvBK+91] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.
- [GAS+00] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33(3), March 2000.
- [GF82] Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–119, January 1982.
- [GG78a] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 231–240, January 1978.
-

-
- [GG78b] Susan L. Graham and R. Steven Glanville. The use of a machine description for compiler code generation. In *Proceedings of the Jerusalem Conference on Information Technology*, pages 509–514, 1978.
- [GG90] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-hall, second edition, 1990.
- [GH84] Susan L. Graham and Robert R. Henry. Machine descriptions for compiler code generation: Experience since JCIT-3. In *Proceedings of the Jerusalem Conference on Information Technology*, pages 236–250, 1984.
- [GHS82] Susan L. Graham, Robert R. Henry, and R. A. Schulman. An experiment in table driven code generation. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 32–43, June 1982.
- [Hen64] F. C. Hennie. Fault detecting experiments for sequential circuits. In *Proceedings of the Fifth Annual Symposium on Switching Theory and Logical Design*, pages 95–110, November 1964.
- [HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.
- [HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [HYHD95] Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David L. Dill. Architecture validation for processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 404–413, 1995.
- [Int93] Intel Corporation. *Pentium Processor User's Manual: Architecture and Programming Manual*, 1993.
- [Joh83] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *UNIX programmer's manual*, pages 353–387. Holt, Rinehart and Winston, 1983.
- [JPARG68] W. L. Johnson, J. H. Porter, S. I. Ackley, and D. T. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, 1968.
- [JR] S. C. Johnson and D. M. Ritchie. The C language calling sequence. Bell Labs.
- [KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [KJ74] Niklaus Wirth Kathleen Jensen. *Pascal User Manual and Report*. Springer-Verlag, second edition, 1974.
-

-
- [Koh78] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, second edition, 1978.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.
- [LCH+80] Bruce W. Leverett, Roderic G. G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz, and William A. Wulf. An overview of the production-quality compiler-compiler project. *IEEE Computer*, 13(8):38–49, August 1980.
- [LS83] M.E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. In *UNIX programmer's manual*, pages 388–400. Holt, Rinehart and Winston, 1983.
- [LSU89] Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Press, 1989.
- [Lun83] Hans Lunell. *Code Generator Writing Systems*. PhD thesis, Linkoping University, Linkoping, Sweden, 1983.
- [Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 35(5):1045–1079, 1955.
- [Mil99] Christopher W. Milner. Pipeline descriptions for retargetable compilers: A decoupled approach. Technical Report CS-99-11, Department of Computer Science, University of Virginia, Charlottesville, VA, June 1999.
- [Mot85] Motorola Corporation. *MC68020 32-Bit Microprocessor User's Manual*, 1985.
- [Mot88] Motorola Corporation. *MC88100 RISC Microprocessor User's Manual*, 1988.
- [Mul93] T. Muller. Employing finite automata for resource scheduling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 12–20, 1993.
- [PF94] Todd A. Proebsting and Christopher W. Fraser. Detecting pipeline structural hazards quickly. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 280–286, 1994.
- [RD98a] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98)*, pages 172–188. Springer Verlag, 1998.
-

-
- [RD98b] Norman Ramsey and Jack W. Davidson. Specifying instructions' semantics using csdl (preliminary report). Technical Report CS-97-31, Department of Computer Science, University of Virginia, Charlottesville, VA, June 1998.
- [RF95] Norman Ramsey and Mary F. Fernández. The new jersey machine-code toolkit. In *1995 Usenix Technical Conference*, pages 289–302, January 1995.
- [RF97] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.
- [SL89] Deepinder P. Sidhu and Ting-Kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions of Software Engineering*, 15(4):413–426, April 1989.
- [Sta92] Richard M. Stallman. *Using and Porting GNU CC (Version 2.0)*. Free Software Foundation, Inc., February 1992.
- [Sun87] Sun Microsystems Corporation. *The SPARC Architecture Manual, Version 7*, 1987.
- [Wha90] David B. Whalley. *Ease: An Environment for Architecture Study and Experimentation*. PhD thesis, Department of Computer Science, University of Virginia, 1990.
- [Wic75] John D. Wick. *Automatic Generation of Assemblers*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1975.
- [YL95] Mihalis Yannakakis and David Lee. Testing finite state machines: Fault detection. *Journal of Computer and System Sciences*, 50:209–227, 1995.
- [ZT00] Cindy Zheng and Carol Thompson. Pa-risc to ia-64: Transparent execution, no recompilation. *IEEE Computer*, 33(3), March 2000.
-