SPARC-V9 ARCHITECTURE SPECIFICATION WITH RAPIDE

Alexandre Santoro

Woosang Park

David Luckham

Technical Report: **CSL-TR-95-677**

(Program Analysis and Verification Group Report No. 71)

September 1995

This research has been supported by DARPA under ONR contract N00014-92-J-1928, contract N00014-93-1-1335 and under TRW, subcontract FZ2394LK1S-04.

SPARC-V9 Architecture Specification with Rapide

Alexandre Santoro   Woosang Park   David Luckham

Technical Report: CSL-TR-95-677
Program Analysis and Verification Group Report No. 71

September 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

This report presents an approach to creating an executable standard for the SPARC-V9 instruction set architecture using Rapide-1.0, a language for modeling and prototyping distributed systems. It describes the desired characteristics of a formal specification of the architecture and shows how Rapide can be used to build a model with these characteristics. This is followed by the description of a simple prototype of the proposed model, and a discussion of the issues involved in building and testing the complete specification (with emphasis on some Rapide-specific features such as constraints, causality and mapping). The report concludes with a brief evaluation of the proposed model and suggestions on future areas of research.

Key Words and Phrases: SPARC-V9, Rapide, instruction set architectures, modeling, formal specification, constraints

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report describes an implementation-independent model of the SPARC-V9 instruction set architecture standard[WG94] using the Rapide prototyping language. The model described here is intended for multiple use. First, it is to be an on-line *executable* manual to help developers understand the standard. Second, it should serve as a design-verification tool, to which actual implementations will be compared in order to verify conformance to the standard. Finally, it is an exploration tool, enabling the user to easily modify the model in order to create different versions of the standard.

An instruction set architecture (ISA) is a description of a processor. It defines the processor as a series of state storage units (the registers), a list of opcodes, explanations of how the opcodes affect the state of the machine, and how it reacts to other forms of input such as interrupts and reset signals. Usually, ISA descriptions concentrate on defining the behavior of the architecture, leaving actual implementation issues (the width of the data bus, for example) to the processor designer.

The SPARC-V9 instruction set architecture specification[WG94] uses plain English, along with diagrams to explain how the instruction set architecture behaves. Even though that is the usual way of explaining things, the English language is, by nature, ambiguous. Some descriptions may be misinterpreted, possibly leading to an erroneous implementation of the architecture. This makes it necessary to come up with a specification that is preferably both unambiguous and easy to understand.

An approach to this problem is to use a formal specification language to describe the standard. By using a formal specification, one avoids ambiguity problems since the language constructs that describe the model are completely specified and their meaning is unambiguous. This should remove any questions or doubts introduced by the use of the English language.

There is, of course, a price to pay for using a formal specification language to describe the standard. First, in order to make it well-defined, it is necessary to restrict the meaning of each language construct. The resulting code will have to be more verbose and complex when describing something, so that understanding it will take more effort. Second, the precise, well-defined behavior of the language makes it hard to implement non-determinism in the model, should it be so desired. This non-determinism contradicts the concept of complete formal specification and the result is that it is not present in the languages most commonly used for describing hardware: Verilog [TM91] and VHDL[VHD87]. This is not to say that non-determinism cannot be emulated; it can, but usually requires complex coding to circumvent the language's limitations.

A third issue related to specification languages is the choice of the domain. One can specify a model through several different domains. In the case of hardware design, for example, the common domains are behavioral, structural and geometric[GT88]. A *behavioral* specification describes a model in terms of what it does; a *structural* specification defines a model in terms of the structures that have to be present and how they interconnect; a *geometric* specification describes a model in terms of the geometric layout of its components. Each has a different use, depending on the objective of the specification. Specification languages are geared towards different domains and the issue is choosing one that is adequate for the correct model description.

Even within a domain, there is still the issue of at which level one wants to look at the model. Suppose one wants to write a hardware model of a processor, describing only its behavior. One can do it by describing the model at the register-transfer level, indicating which operations result in changes to the model's registers; one could use traditional programming language constructs such as *if* and *case* statements to help describe the flow of information. One could also describe the model as a series of logic equations, which would reflect more thoroughly an implementation, but be much more complex to understand. Clearly, it is necessary to choose which level of abstraction is desired and choose a language that favors that choice.

Still, there are advantages to using hardware description languages for describing architectures. A formal specification provides a framework for verifying the correctness of an implementation. It states the expected behavior/structure/layout of the system and it is up to the designer, maybe with the help of some tools, to check if the implementation conforms to the specification. To achieve this goal, many specification languages are accompanied by compilers, browsers and other tools that simplify their use in simulation and verification.

Since instruction set architectures are usually implemented as hardware, they are often described by hardware description/simulation languages, since these languages serve as the framework for building actual implementations. Verilog and VHDL are the most common languages for hardware description. They are powerful and versatile, allowing mixed-mode (containing both structural and behavioral parts) descriptions of a model. Unfortunately, they are strongly geared towards simulation, thus requiring too much detail in their descriptions, and forcing the designer using such a model as the specification to worry about bit-level design and implementation issues much earlier than one should have to.

Another language commonly used for instructions set specification is ISPS[Bar81]. ISPS is a register-level language designed specifically for describing instruction set architectures, and has been used in previous versions of the SPARC instruction set architecture definitions[Int92]. Though it did a good job of describing the standard, at that time it lacked executability. This made it hard to debug and verify ISPS models for correctness.

In order to formally specify the SPARC-V9 instruction set architecture it is necessary to use a language that is both *high-level* and *executable*. This will lead to a model that is verifiable and easy to understand. Rapide, as will be seen in section 1.2, has the necessary characteristics for this.

In the next pages we propose an implementation of the SPARC-V9 instruction set architecture standard using Rapide. The rest of this chapter describes the SPARC-V9 standard and the Rapide Prototyping language. Chapter 2 describes the suggested approach for creating the executable standard, while chapter 3 shows the results of a small exercise in creating the model using Rapide-0.2, a previous version of the language. Chapter 4 comments on the results and uses of the model, and chapter 5 presents our conclusions. Finally, the appendices present the guidelines for coding the model and the code for the interface of some basic types.

1.1 The SPARC-V9 Instruction Set Architecture Standard

SPARC-V9 is an instruction set architecture standard for a RISC style processor[WG94]. Among its major characteristics are 64-bit integers, 64-bit virtual addresses, support for superscalar and multiprocessor implementations (through its instruction set), and fast context-switching. There are approximately 20 control/status registers, 32 double-precision floating-point registers and up to 528 integer registers. There are 154 different instructions.

The reference [WG94] describes the SPARC-V9 standard by first defining some terms in order to provide a common language with the reader. It then goes on to broadly define two units, an integer and a floating point unit. This is followed by a definition of the operations on the architecture, the data and instruction formats it uses, the registers that have to be present and the behavior of instructions and traps, as well as the memory model.

All this explanation is done with as little detail as possible. Arithmetic operations are not defined in any detail, other than saying that they are there; if something causes a state change in a register, the exact mechanism of how the state is changed is not mentioned. Take, for example, the case of memory access[1]. Instead of describing each and every instruction for memory access and how they affect the state of the machine, memory access is described by lumping these instructions together and stating what kinds and sizes of transfers there are; from register to memory, from memory to register, in 8-bit, 16-bit, 32-bit and 64-bit sizes. There are no references, at this point, to the exact format of each instruction, its exact operation or possible side-effects.

More rigorous definitions come with the appendices, with their sets of norms. In them, each instructions is treated separately (or in small groups) in one to three pages. These pages specify the instructions' opcode, the suggested assembly syntax and a more detailed explanation of the behavior. The definition of the LDUW instruction[2], for example, shows the opcode and suggested assembly language syntax for the command, specifies how the instruction works, indicating which registers are affected and even mentioning the required memory alignments. It ends with a list of the possible exceptions that this instruction might

[1] Weaver and Germond, *The Sparc Architecture Manual, Version 9*, page17

[2] Weaver and Germond, *The SPARC Architecture Manual, Version 9*, pages 175-176

cause. All in all, a much more detailed description of the instruction than that found in the body of the text.

It should be noted that, though the manual tries to be as detailed as possible, it never tries to tie the standard to any particular implementation. It suggests only the existence of two loosely defined units, the integer unit and the floating point unit mentioned earlier. These units are defined with respect to the expected state transformations, and no reference is made as to how these transformations are to be accomplished. Instructions, when grouped together at all, are done so solely on the basis of similar functionality. Conditional branches, for example, are grouped together since their behavior only varies in what condition is being tested.

Another aspect that the standard treats by omission is concurrency. It avoids tying in together the behavior of different instructions and, even within one instruction, avoids imposing any order on events. The same LDUW instruction described above, for example, makes no reference as to the order in which the operands that help in computing the effective address are returned; provided that the address is computed correctly, the order in which that is done is not important.

The only attempt at imposing some ordering in the model is made at the inter-instruction level. The standard states that any implementation should behave as if it were a serial model[3]. That is, any implementation should behave as if it executed in *program order*, completing an instruction before starting the next one. How this effect is to be achieved is left open.

So, though the manual does a good job in specifying the architecture, it has some weak points. First, it avoids several issues by describing things at the highest possible level and not getting into necessary detail. Second, it avoids tying in parts of the architecture, unless absolutely necessary. Finally, its use of the English language, though useful for gaining an overall understanding of the model, leaves some details unanswered and provides potential for misinterpretation.

1.2 The Rapide Prototyping Language

In order to satisfactorily implement the executable standard, the language used has to have several important characteristics. First, it should have mechanisms that allow abstraction and encapsulation, so that the user does not have to worry about implementation details. Second, it should be executable so that it is easy to simulate the model in order to verify its correctness and functionality. Finally, it should allow behavioral modeling, since this is how the standard defines the instruction set architecture and avoids implementation-dependent issues. Rapide-1.0 satisfies all these requirements.

Rapide is a programming language framework designed with the objective of making it easy and fast to build prototypes of distributed systems. It is an *event-driven* simulation language with constructs that facilitate the description of concurrency and timing. Among its features there are such object-oriented paradigms as inheritance and opera-

[3]Weaver and Germond, The SPARC Architecture Manual, implementation note, page 61

tor overloading. Rapide consists of four sublanguages: types, constraints, executable and architecture[Tea94d, Tea94c, Tea94b, Tea94a].

A Rapide model consists of a set of *modules* connected by an *architecture* description. Each module has an interface and a body. The interface defines the type of the object, specifying how an object of that type communicates with the rest of the world. The body of a module is specified by module generators, which are pieces of Rapide code implementing the actual behavior of the module. An architecture connects several modules together to form either a model or, in a hierarchical manner, another higher-level module.

Models written in Rapide can be compiled and executed. Part of the model construction consists of defining event-generating *actions.* When these actions are triggered, they generate events which can then be used to trigger other actions, thus executing a simulation. It is important to note at this point that, in reality, patterns of events are used to trigger actions, allowing for very complex designs. Rapide simulations also use Fidge-Mattern vectors[Fid91, Mat88] in order to collect dependency and causality information between events.

The result of executing a model is a file containing a set of events and dependency relations between them. This simulation output, also known as a *partial-order event set (poset)*, can be represented as a directed acyclic graph, where each node corresponds to an event and each directed edge represents a causality relationship between the nodes. For example, figure 1.1 shows a possible result of the simulation of an ADD instruction using a Rapide model. In the figure, each ellipse corresponds to an event while the arrows indicate dependency relations between them. This example shows that an ADD operation consists of several events: decoding the instruction, fetching the data from the register set, adding the operands, and storing the result in the appropriate register. Notice that the poset shows no ordering relationship between the two operand fetches; they can be executed in any relative order, even in parallel. All it shows is that the addition operation requires two operands which must be be obtained before the sum can be computed.

Thus, a poset contains more information than the traditional linear traces provided by most simulators. Actually, it can be said that a poset is a compact representation for *all* valid linear traces corresponding to a simulation. In figure 1.1, for example, the poset represents two possible linear traces: $decode, fetchR1, fetchR2, add, storeR3$ and $decode, fetchR2, fetchR1, add, storeR3$. A poset with n events may correspond to a set of up to $n!$ equivalent linear traces.

There are currently several tools available for analyzing posets. The two most useful ones are, without a doubt, the partial order browser (**pob**) and the Rapide animator (**raptor**). The **pob** allows one to display and analyze the poset interactively. One can easily rearrange how the nodes are displayed, trace the dependency chain of an event, inspect the parameters of a node and much more. The other tool, **raptor**, allows one to get an animation of the simulation and observe the event executions. This makes it a very good tool for answering the question of *what* happened, while the **pob** is more suited for explaining *why* things happened.

Up to now we have described the main characteristics of Rapide as a language for building prototypes. There are two other features of the language, that make it well-suited

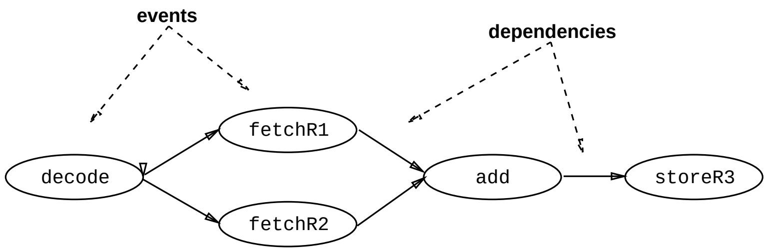

Figure 1.1: Sample resulting poset

for describing instruction set architectures and aiding in design verification: *constraints* and *maps*. These, as the reader will see, are fundamental in making Rapide an useful language for simulation and verification.

A constraint is a description of some behavior, but not its implementation. It describes what one expects to see in a simulation (or maybe what one never wants to see) but does not enforce it to happen. It works as a checker, informing the user of when the behavior deviates from the one specified in the constraint. With Rapide's pattern language, it is possible to describe complex event patterns that one wants to observe. For example,

match ((?addr **in** SPARCint_t, ?val **in** ReadWriteReg_t)
Write(?addr, ?val) -> Read(?addr, ?val)∧(~ *))∧(->*);

is a Rapide constraint specifying register file coherence. It states that when one reads a register, the value returned should be the same as the the last value stored in that register.

One very nice feature of Rapide is that constraint verification is automatic. During a Rapide simulation, new events are continually being checked against the constraints. If some event causes a pattern to match a "never" constraint or not match an "always" constraint, a *constraint violation* event[4] is generated. Using the **pob** tool described earlier, it is very easy to locate these violations and trace them back to their cause. Figure 1.2 shows such a case, corresponding to the constraint described above. The parameters shown for each event are, in order, the register address and the value stored there. The value returned by the second read to register 16 (0xFF) is different than the one last stored on that register (0xAA), so the *inconsistent* event is generated.

Mapping is the second feature of Rapide that makes it a very useful language for verifying conformance to a standard. As was mentioned earlier, the main use of this model is as a standard for design verification. When actual implementations are coded, it should be possible for them to be compared to this model to see if they do indeed conform to the SPARC-V9 standard by obeying the constraints. To do so, it is necessary to be able to get the poset (or linear trace) generated by some other model, and see if it is a valid poset for a SPARC-V9 implementation.

[4]Constraint violation events sometimes are also called *inconsistent* events

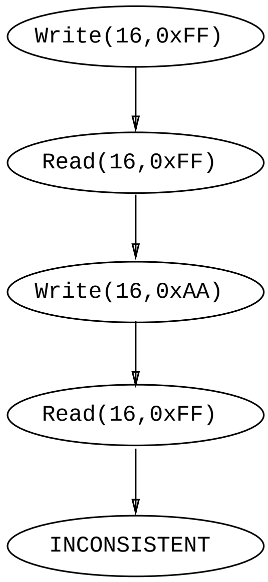

Figure 1.2: Poset indicating a constraint violation

The one problem with this approach is that there might not be a one-to-one correspondence between the events in the implementation and the ones described in the reference model. For example, the implementation being tested might be designed at the gate level, while the model is written at the register transfer level.

Rapide provides a mechanism for dealing with this kind of problem, known as *mapping*. As can be seen in figure 1.3, mapping is a translation of a sequence of events and patterns in one domain to an equivalent event or pattern in another. In this way, a detailed model can be written for a specific implementation, along with a map that would show how the actions in this implementation correspond to actions in the standard. The implementation being verified can then be executed and the resulting poset automatically compared with the reference model.

Constraints and maps make Rapide well-suited for the construction of models for design verification. The ease with which one can describe constraints and the powerful pattern language make it simple to build complex rules. The executability of the model allows one to verify its correctness automatically. Finally, mapping makes it very easy to get an actual implementation and test its behavior.

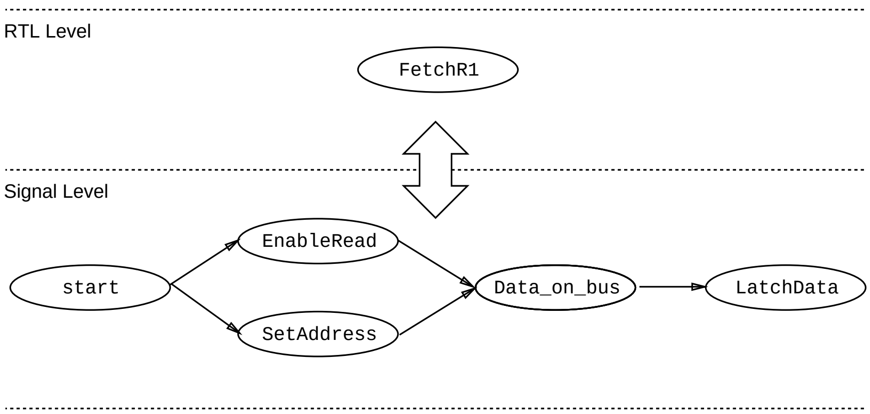

Figure 1.3: Example of mapping

Chapter 2

The Model

2.1 Objectives

Before describing an approach to implementing the SPARC-V9 standard it is necessary to define what the desired characteristics of the model are. By describing what is expected of the final model, these characteristics serve as guidelines for the actual implementation of the standard.

In order to determine what are the model's desired characteristics, it is necessary to understand what is its intended use. Since the model is to be an executable formal specification of an instruction set architecture standard, it has to serve two different purposes: it is a *reference* source, and a *template* against which actual implementations are compared. As a reference source, it is used to explain what makes up the system and how it works. In order to do that it should provide a clear and complete description of what the architecture does so as to leave no room for doubt. As a template, it should simplify the comparison process to actual implementations.

In order to efficiently perform these functions, the model should have the following characteristics:

- correctness
- completeness
- clarity
- precision
- executability
- implementation-independence

Correctness is what assures the user that whatever is seen in the model should actually be there. If something is present in the model, it should correspond to an actual constraint or specification in the English language description. Care should be taken to avoid over-specifying the standard by adding restrictions that are not present in the original documentation.

Completeness guarantees that everything that is in the architecture specification is also in the model. Completeness is, in a way, the dual of correctness. While the latter states that if something is in the model it should also be in the manual, the former states the opposite: if it is in the manual, it should also be in the model. This is, of course, a necessary characteristic of any model, because if some part of the specification is left out, it cannot be said that the model corresponds to the standard.

Clarity is another fundamental characteristic of the model, since it is to be used as a reference source, both in the source and executable code format. The standard should define the operations of the system not only correctly, but also in an easy to understand fashion.

Precision is another of our requirements. While clarity is concerned with how easy it is to read and understand the code and/or simulation results, precision has to do with how well-specified the description is. The manual was written in English, a natural language which allows for misinterpretations and ambiguities. A formal specification should not allow this to happen and must, by necessity, be precise and exact in its statements. Precision should not be confused with determinism; the latter is an implication that the model should always behave in a specific way, while the former just implies that the description should be unequivocal. Thus, it is possible to have a precise (well-defined and understood) construct that is non-deterministic (may take one of several possible behaviors, without the reader being able to determine which beforehand).

Executability is a desired characteristic of the model that adds a totally new dimension to it. By making the model executable it is possible to use it as an on-line tool for queries about the behavior and functionality of the instruction set architecture. It also allows one to have animations of the model's behavior that make it easier to grasp some concepts, and trace executions in order to get the order of events that led to an unexpected (or maybe expected) result.

Implementation-independence is, perhaps, the most important characteristic of the model. This comes directly from the manual, which makes no mention of hardware implementation details. It does not imply in any way that an implementation of the architecture should be pipelined, superscalar or anything; it just presents information on how the state changes in response to specific input to the system. This lack of restrictions as to how actual implementations of the architecture should behave is an intrinsic characteristic of the standard and should be preserved, which makes it necessary for the reference model to be flexible enough so as not to restrict all possible architectural variations one might find in an implementation.

The list of characteristics defined above describe what is expected of the Rapide-1.0 model of the SPARC-V9 instruction set architecture. It is necessary for the model to have all these characteristics, in order to be an adequate tool for answering questions about the standard. This implies that these characteristics should serve as guidelines for how the model is to be built. The next section describes an approach to building the model based on these desired characteristics.

2.2 Approach

In the previous section we defined several characteristics that the model must have in order to be an adequate tool for answering questions about the standard. In this section we present an approach to specifying the SPARC-V9 standard using Rapide-1.0 that, we hope, will fulfill all these requirements. These characteristics will be tackled, one at a time, with methods and solutions being proposed for obtaining the desired goal.

There is, unfortunately, no simple method for guaranteeing completeness and correctness. The only way to attain this goal is by careful programming, attention to detail and extensive testing. There are many software engineering techniques that are of aid in attaining this goal, such as top-down design, modularity of all the model's components, object-oriented techniques, and well-defined interfaces for the several components of the model.

Clarity, in this report's context, means the ability to make the source code readable and the simulation results simple to understand. To attain this goal there are several orthogonal approaches:

- **Model decomposition:** The first thing that can be done is to decompose the model into several components, each dealing with a different aspect of the standard. This will break up the model into several small and less complex units which can be dealt with independently. Since each component can then be studied in isolation, the amount of information that has to be absorbed at a time by the user is much smaller. Rapide has characteristics that make it suitable for this decomposition approach. Its type language[Tea94d] allows not only for the definition of components with well defined interfaces, but also the decoupling of the interface from the implementation of the component's behavior. The issue, then, is how to break down the model. In section 2.4 the suggested decomposition of the model into its constituent components is discussed in more detail.

- **Uniform coding style:** The second approach to dealing with readability is writing the source code in a clear, consistent and uniform style. Such a commonality in coding simplifies the task of understanding the program, since several important attributes of the source code will not change. The way to do this is by coming up with a series of guidelines for writing the code to give it a uniform look. Such guidelines would deal with several aspects of code writing like naming conventions, templates for common pieces of code and the organization of the code itself. Guidelines are presented in appendix A.

- **Library of basic types:** A third way to make the code easier to understand is to define a series of basic, well-understood types that can be shared by all the components. These basic types, implementing things like registers, stacks and 64-bit integers will make the code simpler, since it will not be necessary to reinvent them for each component in which they will be used. Another advantage of using this approach is that it adds another level of abstraction to the model itself, freeing the user from having to worry about details such as how a register is implemented. Section 2.3 suggests a collection of such basic types and presents their interfaces.

It is interesting to note that all three of the above suggestions do more than just improve the clarity. Decomposition, abstraction through the use of basic types and uniformity of code also simplify testing and make it easier to build correct programs.

Precision, as was defined earlier, is intended so as to avoid misinterpretations and ambiguities in understanding the architecture. Most of these arise out of the fact that English, the language used to describe the architecture, is naturally ambiguous. By using a formal specification language such as Rapide-1.0, this requirement is fulfilled automatically. Rapide is completely specified and its constructs have well-defined meanings, so these ambiguities should not appear. Rapide also has the advantage that it allows non-determinism in its constructs, so that any *intentional* non-determinism of the model can be easily implemented.

Executability was another required characteristic of the model, since that would simplify testing of the model for correctness and completeness, as well as provide, through simulation results, a new way to analyze and understand the model. Fortunately, the Rapide specification language comes with tools for compiling and analyzing Rapide models. In order to make the model executable, then, all that is necessary is to construct a top-level architecture connecting all the components of the model and then compile it.

The final, and probably most important characteristic of the model should be implementation-independence. The instruction set architecture manual attains this goal by avoiding overspecifying. Whenever possible, it describes the behavior of some part of the architecture without connecting it in any way to the rest of the standard. The same thing should be done for the executable model. When defining the top-level architecture connecting all the components, care should be taken to avoid tying the model down to a specific implementation. There are many ways that this can be done such as, for example, using generic broadcast buses for communication between the components. By making this architecture as generic as possible, it is possible to avoid adding constraints to the model that were not in the original instruction set architecture.

In order for the model to be satisfactory it has to have the characteristics mentioned in the previous section: correctness, completeness, clarity, precision, executability and implementation-independence. These characteristics define the basic approach to building the model. These involve guidelines for coding, the use of a set of basic types to add a level of abstraction to the model, the break down of the model into several distinct components and the definition of an architecture loosely connecting them. The model should be coded in Rapide, which provides constructs and mechanisms for implementing all these requirements, as well as adding precision and executability.

The next three sections explain in some more detail some of the concepts described in this suggested approach. Section 2.3 describes the basic types used as building blocks for the model, while section 2.4 defines the components that make up the model. Finally, section 2.5 describes the architecture connecting the components in order to form the complete model itself.

2.3 Basic Type Set

A basic type set is a collection of types implementing objects with a desired functionality, that serve as building blocks for building more complex models. From the discussion in the previous section, it should be obvious that there are several advantages in using a basic type set. First of all, they provide basic blocks which can be reused throughout the model, thus saving on development time. Second, they make the code more uniform and thus improve its clarity and precision. Third, they provide a layer of abstraction, freeing the user and/or developer from having to worry about low-level implementation details. These are all desired characteristics of the SPARC-V9 model, and using a basic type set in the model's development is, thus, very useful.

Earlier on it was mentioned that Rapide has the necessary object-oriented paradigms, such as interface/implementation isolation and inheritance. Together with Rapide's behavioral constraint constructs, these facilities make it easy to build basic types with well defined behavior. The question left to be answered is, then, what these basic types should be.

The purpose of the basic type set is to provide the model designer with a homogeneous and consistent collection of types that one can use to build the model. To be useful, this set of basic types must have several properties. First, it must provide the desired level of abstraction so that the designer is not bogged down with details. Second, it must contain type definitions that correspond to hardware entities described in the manual. Finally, it must contain a big enough variety of types so that the user does not have to build new ones, possibly adding errors, just so as to complete the set of useful types.

Reading the SPARC-V9 standard[WG94], one notices that there is only one type of structure defined in the whole text: registers. To be precise, most of the document consists either of register definitions or descriptions of how instructions affect them. This makes a register the natural candidate for the starting point of the type set.

All these registers are not equal. Not only do they vary in size, they also differ in what operations may be performed on them. Some registers, for example, can only be read, while others can also be written to. There are also registers that store data with a specific meaning, such as the integer registers and the floating point registers. This variety suggests that it would be useful to list the main characteristics of each register and use that to group similar registers together. The resulting list is shown in table 2.1, with each group of similar registers separated by horizontal lines.

Table 2.1 suggests that an appropriate approach would be the creation of a register type for each group. Since all groups share some characteristics, the best way to represent them is as a tree, with the root consisting of a base type. New types are derived from the base type by adding new functionality to the type interface. Figure 2.1 shows the resulting type hierarchy. Each box has the type name on top. Inside the box can be found some sample functions and an example of a SPARC register of that type. Arrows go from a supertype to its derived subtype. As can be seen, **ReadReg_t** is the only type that is not part of the hierarchy.

In defining the several register types some conventions have been observed. They are:

Name	Size	READ/WRITE	Operations
r0..r527	64	RW	arithmetic & logic
f0..f31	64	RW	floating point arithmetic
PC	64	RW	sign-extended addition
nPC	64	RW	sign-extended addition
Y	64	RW	
PSTATE	10	RW	
PIL	4	RW	
TBA	64	RW	
CCR	8	RW	
FPRS	3	RW	
FPSR	64	RW	
WSTATE	6	RW	
ASI	8	RW	
TICK	64	RW	increment
TPC	64	RW	push, pop
TNPC	64	RW	push, pop
TSTATE	64	RW	push, pop
TT	64	RW	push, pop
VER	64	R	
CWP	5	RW	increment, decrement
CANSAVE	5	RW	increment, decrement
CANRESTORE	5	RW	increment, decrement
OTHERWIN	5	RW	increment, decrement
CLEANWIN	5	RW	increment, decrement
TL	5	RW	increment, decrement

Table 2.1: SPARC-V9 register set characteristics

Function	Description
[]	Returns the bit value stored at the position specified by the parameter.
field	Returns the field value stored at the position specified by the parameters.
regsize	Returns the size of the register.
intval	Returns the integer value of the register.
sform	Returns the contents of the register in hex form.

Table 2.2: Function description for ReadReg_t

- Bits in a register are ordered from least significant (LSB) to most significant (MSB) in increasing order. The first bit has the index 0.
- Fields are referred to by providing two indices, *index1* and *index2*. The first index is the position in the register of the most significant bit of the field, while the second index is the position of the least significant bit.

The following is the description of the types presented in figure 2.1. Each entry gives a brief description of what the type is like, what is its intended use, and a table with the *added* functionality of each type. We follow the convention that all type names end with '_t'.

One thing that will be noticed in the description is that most of the types have an **intval** function. This function returns an integer representation of the register's contents and its intended use is in debugging, solely as a means to present the value in a form that the programmer can easily read.

The basic types are:

- **ReadReg_t:** This is a read-only register type. It describes the most basic of all register types. As can be seen in figure 2.1, it is the only type among the basic types that is not part of the hierarchy. This happens because, being the only type that cannot be written to, it is inherently different from all other types. It was designed specifically for implementing the VER register. Table 2.2 shows the associated functions.
- **ReadWriteReg_t:** This is the base type for all subsequent types. Its form is very similar to **ReadReg_t**, except that it has been structured so as to allow writing to it. The associated functions are shown in table 2.3. It is intended for the implementation of the basic register structures like PSTATE.
- **UpCounter_t:** This type is a subtype of **ReadWriteReg_t**. It is the first type in which the content of the register has a meaning other than just a collection of bits. In this case, the register type is supposed to store an unsigned integer that can be

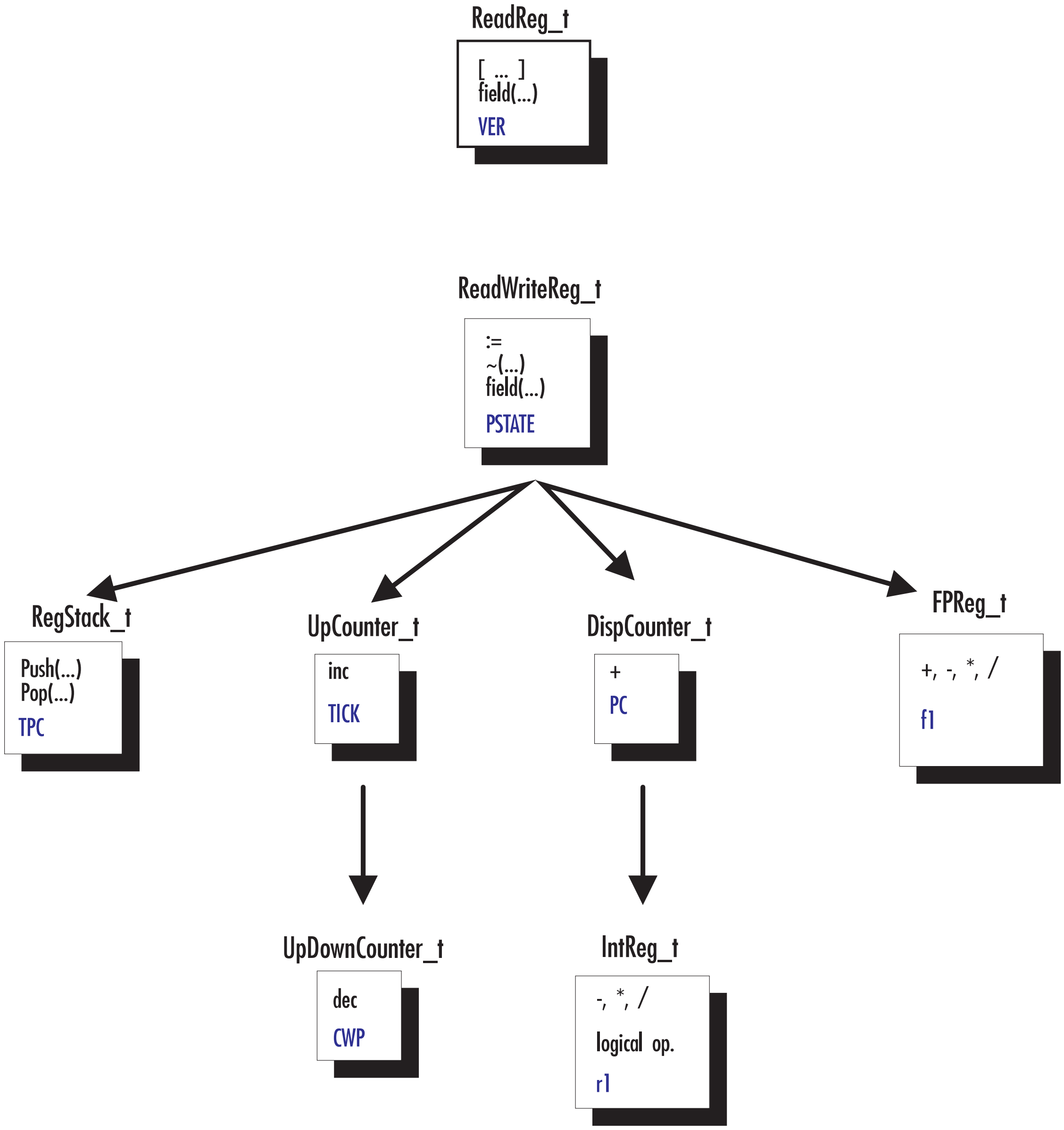

Figure 2.1: Type hierarchy tree for register types

Function	**Description**
[]	Returns the bit referenced by the parameter.
field	Returns the field referenced by the parameters.
~	Toggles all bits in register.
##	Register concatenation operator.
:=	Assigns a value to the register.
regsize	Returns the size of the register.
intval	Returns the integer value of the register.
sform	Returns the contents of the register in hex form.

Table 2.3: Function description for ReadWriteReg_t

Function	**Description**
Reset	Sets the register content to 0.
inc	Adds 1 to the value of the register.

Table 2.4: Added functions for UpCounter_t

incremented. There is no provision for overflow; when the maximum value is exceeded the counter wraps around to 0. It was specifically designed for implementing the TICK register.

- **UpDownCounter_t:** This type is almost the same as **UpCounter_t**, of which it is a subtype. Its functionality is the same, except that it counts both up and down. Again, there is no overflow or underflow; the counter just wraps around. Table 2.5 shows the added functions. It may be used to implement CWP, CANSAVE and the other registers related to register-window control.

- **RegStack_t:** This type is a special type of **ReadWriteReg_t**. As the name implies, it is a stack. Writing to it does not make the previous value disappear, it only pushes it further into the stack. This type was created for implementing the stack-like structures used in trap handling, such as the TT and TPC registers.

Function	**Description**
dec	Decreases the value stored in the counter by 1.

Table 2.5: Added functions for UpDownCounter_t

Function	Description
push	Adds an element to the top of the stack.
pop	Removes the top element from the stack.
top	Returns the value of the top element from the stack.

Table 2.6: Added functions for RegStack_t

Function	Description
+	Adds a sign-extended value to the contents of the register.

Table 2.7: Added functions for DispCounter_t

- **DispCounter_t:** This type stores an unsigned integer to which a displacement value may be added. This displacement value is supposed to be in two's complement format and is sign-extended for the operation. Note that there is no overflow or underflow, just wraparound. It is intended for the implementation of PC and nPC.
- **SPARCint_t:** This type is the subtype of the **ReadWriteReg_t** type, and implements a 64-bit integer. Its intended use is in the 64-bit integer arithmetic module. As such, you can perform all integer arithmetic functions with them, as well as the standard logical ones. All arithmetic operations assume that the integers are in two's complement format.
- **SPARCreal_t:** This is the subtype of **ReadWriteReg_t** intended for the implementation of the floating point unit. It is analogous to the **SPARCint_t** type, but deals with floating point arithmetic instead. It follows the IEEE standard for floating point representation and operations mentioned in section 2.4. The basic arithmetic and comparison operations are provided.

The types presented above are all variations of the basic concept of a register. They vary in the functionality provided, which in turn is a result of the functionality required of the registers defined in the SPARC-V9 instruction set architecture manual. This one-to-one correspondence ensures that the basic type set presented here is more than adequate for building the SPARC-V9 model with the desired level of abstraction. Appendix B presents the code for the basic type interfaces.

2.4 Components

As suggested in section 2.2, it is interesting to break down the model into several distinct components, each one dealing with a different aspect of the instruction set architecture.

Function	Description
&	Bit-wise logical *and*.
or	Bit-wise logical *or*.
xor	Bit-wise logical *xor*.
neg	Two's complement negation.
+	Two's complement integer addition.
−	Two's complement integer subtraction.
*	Two's complement integer multiplication.
/	Two's complement integer division.
mod	Two's complement integer remainder.
shr	Shift right operator.
shl	Shift left operator.
ashiftr	arithmetic shift right operator.
=	Equality comparison operator.
/=	Inequality comparison operator.
>	Greater-than comparison operator.
<	Smaller-than comparison operator.
>=	Greater-than-or-equal comparison operator.
<=	Smaller-than-or-equal comparison operator.

Table 2.8: Added functions for SPARCint_t

Function	Description
+	Floating point addition.
−	Floating point subtraction.
*	Floating point multiplication.
/	Floating point division.
sqrt	Floating point square root.
=	Equality comparison operator.
/=	Inequality comparison operator.
>	Greater-than comparison operator.
<	Smaller-than comparison operator.
>=	Greater-than-or-equal comparison operator.
<=	Smaller-than-or-equal comparison operator.

Table 2.9: Added functions for SPARCReal_t

This decomposition allows one to study, debug and test the model one piece at a time, which reduces the complexity of the problem and improves the model's correctness. It just remains to be decided how this decomposition should happen.

One problem is that there are many possible decompositions of the model, all equally valid in that they result in a working model. Choosing between them, then, is a matter of functionality, i.e., finding an organization that reflects the kind of information one wishes to convey.

A first step in deciding how to decompose the units is to define what are the main characteristics of any component. Basically, components can be fit into two broad categories: *state units* and *execution units*. State units are those that consist of sets of registers that store the state of the system, such as the integer registers, the program counter, etc. The execution units are those responsible for actually performing operations that cause the state of the model to change.

The SPARC-V9 architecture defines both state that has to be stored and operations that change the state. Our suggested approach is, therefore, to divide the system into both state and execution units. Having just one state unit and one execution unit, though, still result in unmanageable chunks of code, so further decomposition is necessary. Here again the English manual suggest a good approach; in its description of the opcodes, it groups similar instructions together and we will borrow that concept.

This suggests the following decomposition for the units (all references are to chapter 5 of [WG94]):

- State units decomposition:

  control/status registers: This unit contains all registers associated with processor status and control such as the program counter, nPC, PSTATE, etc. Most, but not all, of the control/status registers would be placed here. Those that are intimately associated with some specific function, such as the register window control registers, will be placed elsewhere.

  integer register set: This unit contains the integer register set, as well as the register window state registers defined in section 5.2.10 of [WG94]. Since the actual size of the integer register set is implementation-dependent, the suggested approach is to use a parameterized module generator with the desired number of register windows as the parameter. This way, any possible integer register set implementation can easily be created. The behavior of this unit is quite simple, consisting exclusively of actions for reading and writing its registers.

  floating point register set: This register set contains the 32 single-precision / 32 double-precision / 16 quad-precision floating point registers defined in section 5.1.4 of [WG94] Care should be taken to implement the overlapping, or aliasing, of registers defined by the standard. Like the previous two register sets, the only actions associated with this unit consists of either reading or writing to it.

- Execution units decomposition:

instruction fetch: This unit is responsible for triggering the execution of an instruction. When activated, either by a *start* event[1] or by the completion of the last operation, it accesses the program counter and initiates the fetching of the next opcode from the outside environment. Once the opcode is obtained, it is made available to all execution units in the model.

memory access: This unit, as the name implies, is responsible for controlling the communication between the SPARC CPU and the environment. Like the instruction fetch unit described before, it is not an opcode executing unit. Instead, it provides a service for all other execution units, through which they can access the main memory.

interrupt/trap handling: This is another unit that is responsible for treating two distinct, but related types of events. As an interrupt handler, it observes events coming from the CPU environment (such as a RESET) and takes the appropriate action. As a trap handler, it deals with all internally generated traps (such as overflow, software reset, etc.).

data movement: This unit is responsible for all opcodes related to moving data from one place to another within the architecture. It deals with such simple moves as from one register to another, as well as with loads/stores from/to main memory. In order to perform its function it will have to work closely with the memory access unit.

integer logic and arithmetic: This unit implements a 64-bit arithmetic and logic unit. It performs all the integer operations defined on the standard, on both 32 and 64-bit data. It does not contain any registers for storing data, using instead immediate data passed along with the opcode and/or data from the integer register set.

floating point arithmetic: This unit is analogous to the integer logic and arithmetic unit, the difference being that it operates on the floating point register set. It implements operations such as FADD, FMUL and FDIV, and would consist of a 128/64/32 (quad, double and single precision, respectively) bit floating point ALU conforming to the IEEE 754 binary floating-point arithmetic standard[IEE85].

branch control: The branch control unit deals with all program flow related opcodes. As such it processes commands like JMPL, BPcc and RETURN. It must, by necessity, be able to access and change the value of nPC.

privileged operations: This is the unit that deals with all the privileged instructions. These instructions usually consists of reading and/or writing a set of control registers that change the state of the CPU. Some examples of privileged operations are RDPR, RDTICK and LDDA.

miscellaneous: This unit would group together any opcodes that do not fit into any of the units above. It could, for example, deal with such opcodes as MEMBAR, SIGM and WRCCR.

[1] Start events are events that are automatically generated in a Rapide program when an object is created. They are used, as the name suggests, to start the simulation

The memory access unit and the instruction fetch unit are special, in that they are not directly related to any opcodes, but rather serve as *auxiliary* execution units for the model. Their job is to isolate the model from the environment and provide it with the instructions for execution. The rest of the execution units, on the other hand, are directly connected to SPARC-V9 opcodes. Each opcode is processed totally by one of these execution units (which may have to access some or all of the state units in order to do this), and thus, there is a one-to-one correspondence between execution units and opcodes. This is shown in table 2.10.

It should be emphasized at this point that this is a suggested organization, not a mandatory one. Many other equally good arrangements are possible. For example, both the memory control unit and the interrupt/trap handling unit could be broken up into smaller independent units. Another possible approach would be not to isolate the state from the execution, but keep related operations and registers in the same unit (i.e., have the trap registers inside the trap handling unit).

2.5 Architecture

The previous section described the units that make up the model. The next and final step is to define an architecture connecting these modules. In this case, we follow the definition of architecture proposed in [LVM95], in that an architecture consists not only of components and their interconnections, but also includes protocols defining how these components communicate. This should provide a complete description of the instruction set architecture, not leaving any ambiguities.

The definition of the architecture is a crucial part of the model's design, since that is the point at which the model is usually bound to one specific implementation. In this case, it is desirable to make this binding as loose as possible, so that the model remains implementation-independent. The instruction set architecture manual achieves this goal by avoiding defining any binding architecture at all; instead it keeps the opcode definitions loosely coupled, describing how the architecture reacts to events without tying one event to another.

This approach is, unfortunately, not one that can be used in the definition of the model's architecture. One of the model's objectives is that it is to be used as an *executable* standard and to attain this objective an architecture must be completely specified. The challenge is, then to make this specification as flexible as possible.

The best way to attain this goal is by making the communication mechanism between modules as generic as possible. Instead of defining point-to-point communication between the components, broadcasting should be used. By defining several specialized broadcast channels to which a component can connect, an executable model can be built that retains most, if not all, of the intended implementation-independence and flexibility.

There are many possible such broadcast channel organizations, ranging from "one channel for everything" to the equivalent of point to point connections between modules. Neither of the extreme solutions are ideal: the former complicates the interface of the components,

unit	opcode			
Integer Arithmetic	ADD,	ADDcc,	ADDC,	ADDCcc,
	AND,	ANDcc,	ANDN,	ANDNcc,
	MULScc,	MULX,	OR,	ORcc,
	ORN,	ORNcc,	POPC,	SDIV,
	SDIVcc,	SDIVX,	SETHI,	SLL,
	SLLX,	SMUL,	SMULcc,	SRA,
	SRAX,	SRL,	SRLX,	SUB,
	SUBcc,	SUBC,	SUBCcc,	TADDcc,
	TADDccTV,	TSUBcc,	TSUBccTV,	UDIV,
	UDIVcc,	UMUL,	UMULcc,	XNOR,
	XNORcc,	XOR,	XORcc	
Branch Control	BPcc,	Bicc,	BPr,	CALL,
	FBfcc,	FBPfcc,	JMPL,	RETURN
FP Arithmetic	FABS,	FADD,	FCMP,	FCMPE,
	FDIV,	FdMULq,	FiTO(s,d,q),	FsMULd,
	FSQRT,	F(s,d,q)TOi,	F(s,d,q)TOx,	FMUL,
	FNEG,	F(s,d,q)TOx,	FSUB,	FxTO(s,d,q),
	F(s,d,q)TO(s,d,q)			
Data Movement	CASA,	CASXA,	FLUSHW,	FMOV,
	FMOVcc,	FMOVr,	LDD,	LDDA,
	LDDF,	lDDFA,	LDF,	LDFA,
	LDFSR,	LDQF,	LDQFA,	LDSB,
	LDSBA,	LDSH,	LDSHA,	LDSTUB,
	LDSTUBA,	LDSW,	LDSWA,	LDUB,
	LDUBA,	LDUH,	LDUHA,	LDUW,
	LDUWA,	LDX,	LDXA,	LDXFSR,
	MOV,	MOVr,	MOVcc,	RDASI,
	RDASR,	RDCCR,	RDFPRS,	RDPC,
	RESTORE,	RDY,	RDTICK,	SAVE,
	STD,	STDA,	STDF,	STDFA,
	STB,	STBA,	STF,	STFA,
	STFSR,	STH,	STHA,	STQF,
	STQFA,	STW,	STWA,	STX,
	STXA,	STXFSR,	SWAP,	SWAPA,
	WRASI,	WRASR,	WRCCR,	WRFPRS,
	WRPR,	WRY		
Miscellaneous	FLUSH,	IMPDEP1,	IMPDEP2,	MEMBAR,
	NOP,	SIGM,	STBAR	
Priv. Operations	SAVED	RDPR	PREFETCH,	PREFETCHA,
	RESTORED			
Trap Handling	DONE,	ILLTRAP,	RETRY,	Tcc.

Table 2.10: Execution units and their related opcodes.

while the latter removes all flexibility from the model. Clearly, a middle-ground approach would be best. By keeping in mind that the model consists of execution units responsible for changing state and state units responsible exclusively for storing state, the authors considered the following bus partition to be adequate:

- **control/status bus:** Connects the control/status unit to all the execution units that might want to access it.
- **integer bus:** Connects the integer register set to all units that might want to access it.
- **floating point bus:** Connects the floating point unit to all units that might want to access it.
- **memory access bus:** Connects the memory access unit to all units that might want to access the main memory.
- **instruction bus:** Connects all units that might want to either issue or execute an instruction.
- **acknowledge bus:** Connects all units that might want to signal or be aware than an instruction has been executed.

This bus partition allows for a detailed enough description so that the model is manageable and understandable, while not adding any implementation dependence.

Having determined what the necessary buses are, the next step is to decide which components should be in the model. The suggested approach for its implementation in Rapide-1.0 is a minimalist one. There should be no more than one instance of each type of unit. One of each type of the units described in section 2.4 is necessary, otherwise it would not be possible to build the model. More than one would make some coordination effort between them necessary, and that would add unwanted implementation dependencies and constraints to the model.

Figure 2.2 shows the architecture of the proposed system. The units are represented by rectangles, with the execution units being white while the state units are shaded gray. The vertical lines represent the communication buses described above, while the horizontal ones represent the connections between the units and the buses, with the arrows indicating the direction of the flow of information.

Once the components making up the model and the interconnection mechanism between them is defined, the next step is to define the protocol for the model's top-level behavior. This protocol describes how the model should deal with the instruction "fetch-and-execute" cycle, and should be generic enough so as to avoid adding any implementation dependencies. Based on the broadcasting principles and buses defined in the previous paragraphs, the behavior of the system is the following:

1. The instruction-fetch unit retrieves the current PC value from the control/status register set and uses that to issue a memory access request.

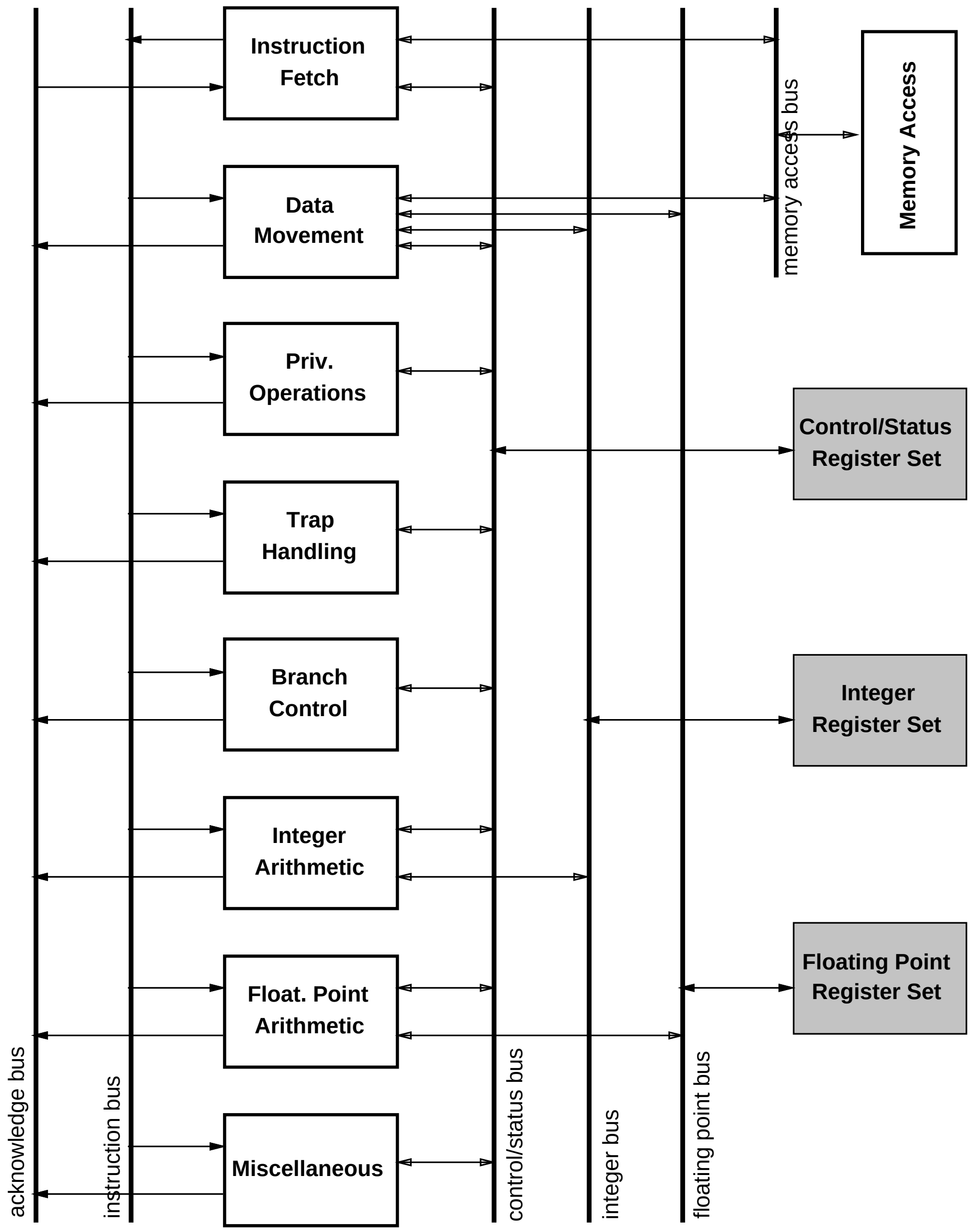

Figure 2.2: Simplified SPARC-V9 Architecture

2. The memory access unit performs the required access and returns the opcode to the instruction fetch unit.

3. The instruction fetch unit broadcasts the opcode to all execution units.

4. The execution unit that has this opcode in its domain decodes it and executes the required operation. This may involve exchanging information with the control/status register units and/or the integer and floating point register sets.

5. The execution unit signals the end of the opcode execution to the instruction fetch unit, indicating that a new instruction cycle may begin.

This protocol preserves implementation-independence by loosely binding the individual components' behaviors. The instruction fetch unit does not care where the instruction is going to, and the execution units do not care where it comes from or if any other unit is also processing it. This adds flexibility, in that it allows for the actual computation to be distributed in several different ways.

Another major characteristic of this model is that it is *sequential* in nature, just like the model described in in the instruction set architecture manual. This serialization, though, is done entirely in the instruction fetch unit and is invisible to any other component of the system. By changing the instruction fetch unit it is quite easy to create other implementations of the architecture, using a pipelined, superscalar or some other approach. Since this is all done inside the instruction fetch unit, it is transparent to anyone looking at the top-level model.

This chapter described what an implementation-independent model of the SPARC-V9 instruction set architecture should look like. It described the desired characteristics of such a model and how one should go about building a model that satisfies these requirements. It then described a basic type set to use as building blocks for constructing the model, as well as the major components that the model should have. Finally, it described an architecture for connecting these components in an implementation-independent way.

Rapide, as a prototyping and simulation language, has all the necessary constructs for implementing the model as described here. It allows the designer to break down the model into distinct components, connect them in the desired way and provides inheritance and polymorphism mechanisms for creating the basic types.

Chapter 3

Scaled-Down Rapide-0.2 Model

Before attempting to build the complete SPARC-V9 model, it is interesting to do a small scale experiment, in order to check the validity of the concepts presented in the previous chapter. This prototype can be used to test the broadcast-bus architecture concept, provide an estimate of its flexibility, and give an idea of what kind of results one can expect from such a model.

The language used for implementing this scaled-down version of the SPARC-V9 is Rapide-0.2[Bry92]. Rapide-0.2, a preliminary "proof of concept" version of Rapide. It was chosen because, at that time the model was built, Rapide-1.0 was still in its early design stages while the 0.2 compiler was complete and stable. Since both Rapide-0.2 and Rapide-1.0 share the same philosophy and language constructs, using the former to create this prototype in no way invalidates the result.

The rest of this chapter describes the scaled-down SPARC-V9 model. Section 3.1 describes the architecture of the model chosen for implementation, while sections 3.2 and 3.3 respectively describe the results of a simulation of a sequential and a pipelined implementation.

3.1 Model Architecture

The objective of this exercise was to build a prototype to test if the approach suggested in the previous chapter is valid. To do so, the model must satisfy two requirements. First, it has to be complete enough so that it can actually represent the SPARC-V9 architecture described in the preceding chapter. On the other hand, it has to be simple enough so that the designer can build it in a short amount of time. After all, the objective of this exercise is only to verify if the approach is feasible, not to construct a full implementation.

Clearly, these two requirements contradict each other; the former implies the use of as many components as possible (to closer approximate the suggested architecture), while the latter tries to minimize the number of units so as to keep the model simple. A compromise must be attained in order to satisfy both of these goals.

The main characteristic of the suggested architecture is the principle of broadcasting instructions and request-acknowledge pairs through a set of buses. This implies that the prototype should have at least two units on the receiving end of any broadcast, otherwise the architecture would correspond to a point-to-point connection and the broadcasting feature would not be tested. This suggests that there should be at least two opcode execution units in the prototype, executing two distinct opcodes. The authors arbitrarily chose ADD and LOAD as the opcodes for testing.

In order to be able to execute these instructions, the model will require some other units, either for interacting with the environment or just for storing information. This leads to the following units being present in the prototype:

- **instruction_fetch unit:** This unit is necessary in any model of the SPARC-V9. It is the unit responsible for coordinating the behavior of the model and providing it with a "heartbeat." It is by changing the behavior of this unit that one is able to create several different implementations, as will be seen later.
- **memory_access unit:** This unit, like the previous one, is necessary for any implementation of the SPARC-V9 standard. The standard is based on the principle that instructions are fetched from memory for execution and without this unit such a basic task could not be accomplished.
- **data_movement unit:** From table 2.10 in section 2.4, it is clear that this unit is necessary for implementing the LOAD instruction.
- **integer_arithmetic unit:** Again, table 2.10 indicates that this unit is necessary for the implementation of the ADD operation.
- **control/status_register set:** This unit is necessary because it contains, among other registers, the program counter, which is used by the instruction_fetch unit to load the next instruction from memory.
- **integer_register set:** The ADD and LOAD operations require operands from the integer register set, and possibly cause changes to the state of one or more of these registers. This makes the presence of the integer_register set necessary.

The units mentioned above are the ones that make up the scaled-down model itself. Other than that, only one more unit is necessary, a **handler unit**, responsible for the job of representing the environment surrounding the prototype, such as the memory and the user interface.

Figure 3.1 shows the detailed architecture of the scaled-down SPARC-V9 prototype in the Rapide-0.2 system, excluding the handler unit. Each *execution* or *status* unit is represented by a rectangle and the *communication channels* are represented by the lines connecting modules. Each unit contains a series of *in* and *out* actions, (the incoming and outgoing *pen* symbols), representing its communication mechanism with the outside world. Each action has a specific function, described in table 3.1.

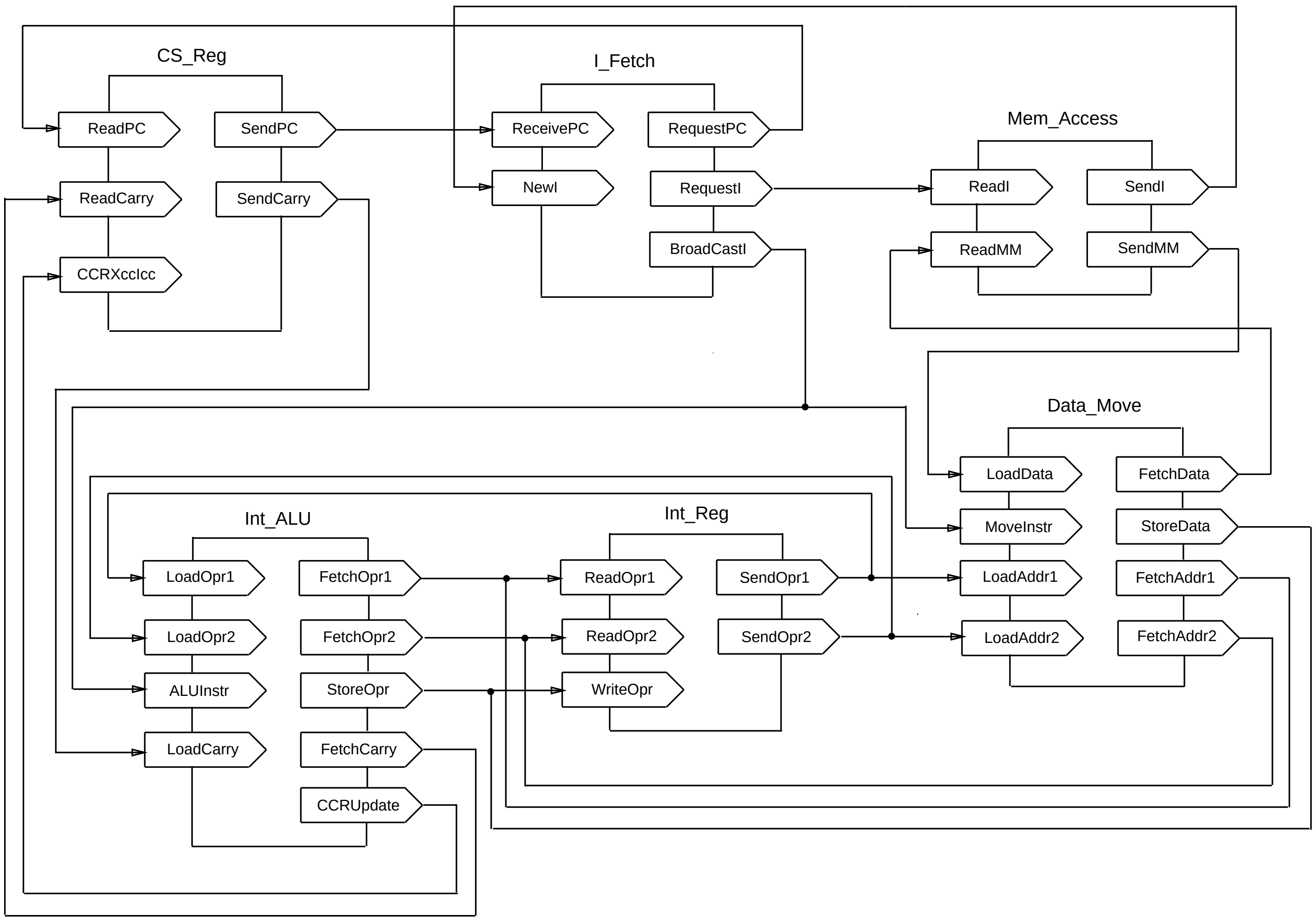

Figure 3.1: Scaled-down SPARC architecture in Rapide-0.2

module	action	type	description
I_Fetch	RequestPC	out	Requests the current value of the PC.
	RequestI	out	Requests the next instruction.
	BroadcastI	out	Broadcasts instructions to other execution units.
	ReceivePC	in	Retrieves the current value of the PC.
	NewI	in	Retrieves current instruction.
CS_Reg	SendPC	out	Broadcasts the current value of the PC.
	SendCarry	out	Broadcasts the current value of the carry flag.
	ReadPC	in	Receives requests for the value of the PC.
	ReadCarry	in	Receives requests for the value of the carry flag.
	CCRXccIcc	in	Updates the current value of the carry flag.
Mem_Access	SendI	out	Broadcasts the current instruction.
	SendMM	out	Broadcasts the current data from memory.
	ReadI	in	Receives address for requested instruction.
	ReadMM	in	Receives address for requested data.
Int_ALU	FetchOpr1	out	Requests operand 1 of arithmetic operation.
	FetchOpr2	out	Requests operand 2 of arithmetic operation.
	StoreOpr	out	Stores result of arithmetic operation.
	CCRUpdate	out	Broadcasts new value of the carry flag.
	AddDone	out	Signals end of operation.
	LoadOpr1	in	Receives operand 1 of arithmetic operation.
	LoadOpr2	in	Receives operand 2 of arithmetic operation.
	ALUInstr	in	Receives opcode for execution.
	LoadCarry	in	Receives current value of the carry flag.
Int_Reg	SendOpr1	out	Broadcasts the value of operand 1.
	SendOpr2	out	Broadcasts the value of operand 2.
	ReadOpr1	in	Receives requests for operand 1.
	ReadOpr2	in	Receives requests for operand 2.
	WriteOpr	in	Receives request for writing to a register.
Data_Move	FetchData	out	Broadcasts request for data from memory.
	FetchAddr1	out	Requests first part of effective address.
	FetchAddr2	out	Requests second part of effective address.
	StoreData	out	Sends result to Int_Reg for storage.
	LoadData	in	Receives requested data from memory.
	MoveInstr	in	Receives opcode for execution.
	LoadAddr1	in	Receives first part of effective address.
	LoadAddr2	in	Receives second part of effective address.

Table 3.1: action definitions for SPARC-V9 prototype

The set of units described above require the presence of all but one of the broadcast buses previously defined. The one that is not necessary is the **floating point bus**, since no floating point operations are performed. Its absence does not invalidate the prototype, since it is analogous to the integer bus in form and function. If the integer bus behaves according to expectations, so should the floating point bus.

This model was designed to allow the user to load a program into the prototype's *main memory* and have it executed. This loading operation, as well as handling the interaction between the user and the model, is done by the handler unit.

In order to study how flexible the architecture and language are, two scaled-down models were created. The first one is sequential in nature and follows the approach described in the previous chapter. The second is pipelined and shows how simple it is to modify the original architecture in order to obtain different implementations.

In the following two sections, we will show the results of running these models. They show typical computer execution cycles: *instruction-fetch, decode, operand-fetch, execute* and *operand-store*. Each run generated a poset, which was analyzed using the **pob**, the graphic browser tool described in section 1.2.

3.2 Sequential Execution in Rapide-0.2

The first model built and tested was the sequential one, since it is the one that is the closest approximation of the architecture suggested in the previous chapter. After it was ready, a program consisting of an ADDcc instruction followed by a LOAD instruction was loaded into the model and a simulation was run. Figure 3.2 shows the poset of the resulting computation. In the figure, each node represents an *event* and each directed edge represents a *dependency* or *causality* relationship between nodes (an arrow going from event A to event B implies that A causes B). The time axis[1] flows vertically, with events that happened later in time appearing below those that happened before. Events happening at the same time are all shown in the same row.

Even though this is quite a simple example, there is much information that can be obtained from it. This is due not only to the nature of the model, but also because of Rapide's properties: the causality and dependency relations contained in the simulation add another dimension of information to the linear traces found in traditional simulators.

First, the figure clearly shows the sequence of events that happen during the execution of an ADDcc followed by a LOAD, specifying what events have to happen, and in what order. From the figure one sees that:

1. The *instruction_fetch* unit requests and receives the current PC value from the *control/status* unit (indicated by the *RequestPC* event).

2. The *instruction_fetch* unit requests and receives the desired instruction from the main memory (indicated by the *RequestI* event), obtained through the *memory_access* unit.

[1]The time units necessary to complete an operation, shown in the top right corner of figure 3.2 is arbitrary

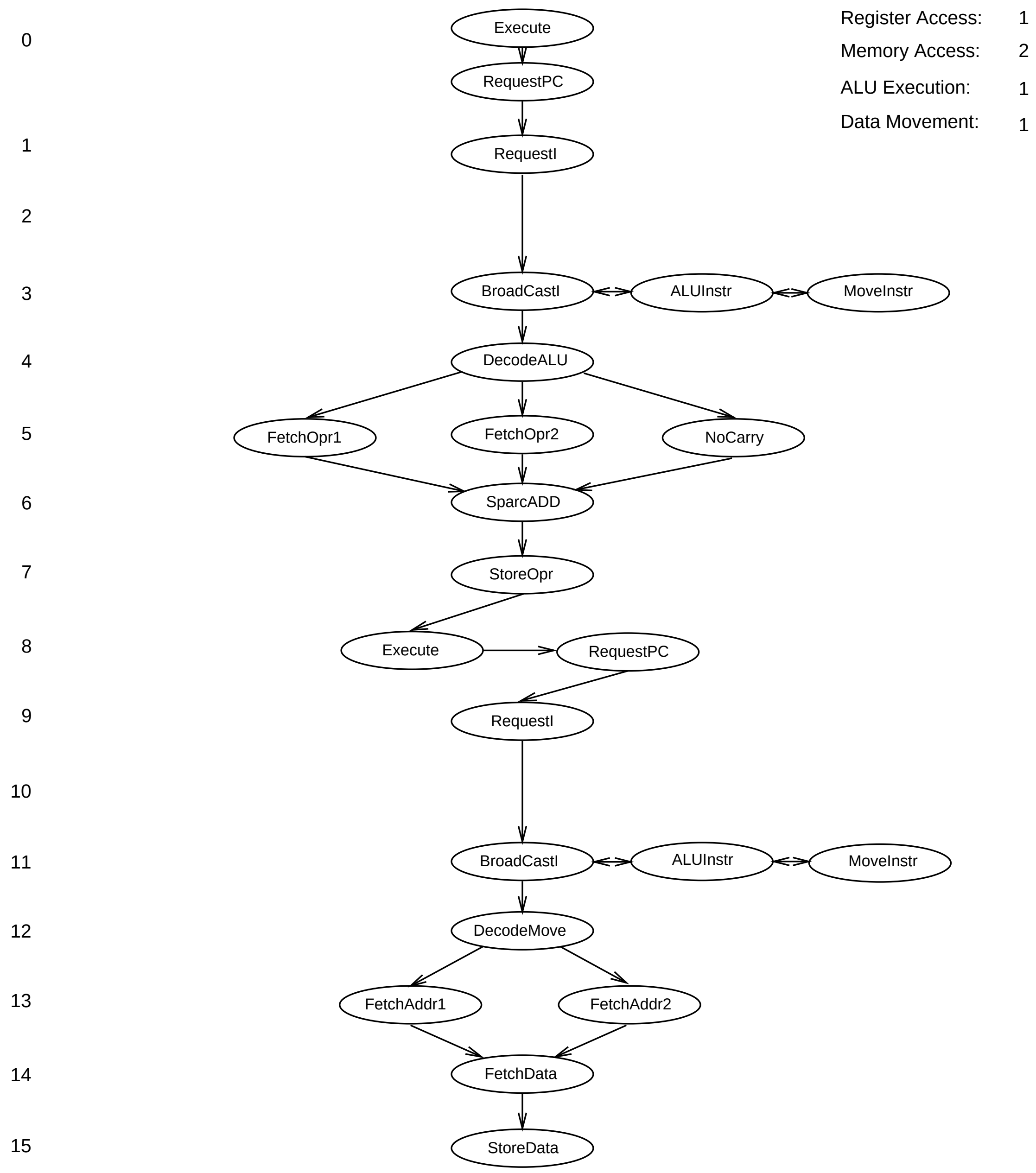

Figure 3.2: Poset of Add-Load Execution in Rapide-0.2

3. Once the *instruction_fetch* unit obtains a new instruction, it broadcasts the instruction to all other execution units through the *BroadCastI* event, making the new instruction available to them.

4. The proper unit (the *integer_arithmetic* unit in this case) decodes the instruction (indicated by the *DecodeALU* event).

5. The *integer_arithmetic* unit requests and obtains the necessary data through the *FetchOpr1, FetchOpr2* and *NoCarry* events[2].

6. The *integer_arithmetic* unit performs the actual addition (indicated by the *SparcADD* event).

7. The *integer_arithmetic* unit stores the result back in the *integer_register set* (indicated by the *StoreOpr* event).

This ends the execution of the first instruction. It is immediately followed by the execution of the LOAD instruction, which has the following format:

1. The *instruction_fetch* unit requests and receives the current PC value from the *control/status* unit (indicated by the *RequestPC* event).

2. The *instruction_fetch* unit requests and receives the instruction from the main memory (indicated by the *RequestI* event), obtained through the *memory_access* unit.

3. Once the *instruction_fetch* unit obtains a new instruction, it broadcasts the instruction to all other execution units through the *BroadCastI* event, making the new instruction available to them.

4. The proper unit (the *data_movement* unit) decodes the instruction, indicated by the *DecodeMove* event.

5. The *data_movement* unit fetches the proper data values, indicated by the *FetchAddr1* and *FetchAddr2* events.

6. The *data_movement* unit fetches the data from memory (indicated by the *FetchData* event), using the *memory_access* unit.

7. The *data_movement* unit sends the resulting data to the *integer_register set*, indicated by the *StoreData* event.

At this point it should be pointed out that the poset shown is not the entire poset generated by the simulation. Some of the events have been removed in order to simplify the visualization of the results. The **pob** graphical browser tool allows one to easily accomplish this task, as well as determining which units generated or received the events, and even to look at the parameters associated with each event.

[2]The *NoCarry* event is necessary due to limitations in Rapide-0.2's pattern language. This limitation no longer exists in Rapide-1.0

A second interesting point about the poset is that, unlike linear traces, it can be used to identify potential parallelism in the model. In figure 3.2 one notices that once a *DecodeALU* event of an ADD instruction has been performed in the *integer_arithmetic* unit, the following events, *FetchOpr1, FetchOpr2* and *NoCarry*, can be performed in any order, since there is no causal relationship between them. These execution flow is *synchronized* again at the *SparcADD* event later on, since it depends on all these three events. This means that these data fetches could have been carried out in sequence, implying a one-port register file, or in parallel, implying a multi-port register file. In this example, the potential parallelism between fetches can represent 6 conventional linear traces. In this way, implementation-independence is maintained, since the resulting simulation is not bound to any specific model.

The sequential model presented in this section has shown that the suggested approach is feasible and has the potential for not tying the model down to any specific implementation, as was seen by the fact that it did not constrain the *integer_register* set to have one or two read ports.

The next section describes a pipelined implementation of the sequential model presented in this section. This way, it is possible to determine how flexible and adaptable the suggested approach might be, as well as what additional information might be obtained from posets.

3.3 Pipelined Execution in Rapide-0.2

The sequential model served to show the validity of the approach suggested in the previous chapter. It was able to prove that the architecture presented works and gave an idea of what kind of results can be obtained from the model. The only question that remains to be answered is whether such an architecture is flexible enough to serve as a the starting point for other implementations.

To answer this question, the authors decided to create a pipelined version of the model. The pipelined model created was simple, and the only modification consisted of changing the *instruction_fetch* unit so that it would start fetching the next instruction immediately after having broadcast the first, instead of waiting for the completion signal[3] event (through the *acknowledge bus*) from one of the other execution units.

In the actual code, this modification consisted of changing just one line of the source code. The original sequential model had the following line in the *instruction_fetch* unit, describing when to start a new instruction:

when Complete **do** Execute;

This line basically says that whenever the instruction_fetch unit receives a *Complete* event, signalling that the processing of the last issued instruction is complete, it should generate a new *Execute* event, starting the processing cycle again.

[3] *StoreOpr* in the ADD instruction and *StoreData* in the LOAD instruction were used as the *completion* events in the previous example.

In the pipelined model, this line was modified to:

when BroadcastI **do** Execute;

This modification ensures that the model does not wait until the processing of the previous instruction is complete. Instead, as soon as the *BroadcastI* event is generated (indicating that the instruction_fetch unit has finished retrieving the instruction and has sent it along its way), the unit immediately starts fetching another instruction.

This approach does not deal with the problem of data dependency between instructions, since that would be time consuming work, and unnecessary for verifying if the modified example works. The solution, though, is not complicated. All that would be necessary would be for the *instruction_fetch* unit to decode the instruction and identify which registers, if any, have to be written to. When another instruction wants to read one such register, the *instruction_fetch* unit can then stall the pipeline until it guarantees that the desired register has been updated.

Once the model was modified and compiled, the authors ran an example consisting of three ADD instructions without any data dependency between them. The result of the simulation is shown in figure 3.3. The figure represents the result in much of the same way as before, with nodes representing events and lines the causality relationship between them. Again, time flows vertically, with events in the same row happening simultaneously, while events in lower rows happen later in time. In order to reduce the complexity of the example some of the events are not shown, as well as the parameters of each event. These, though, can be easily accessed using the **pob** poset analysis program.

Looking at the figure one immediately recognizes three sets of events, connected by edges going from the *BroadcastI* node of one group to the *RequestPC node* of the other group. Each of these sets corresponds, respectively, to the execution of the ADDC, ADDCcc, and ADDcc instructions. Like the one linear ADDcc instruction in the previous example, the behavior is the following:

1. The *instruction_fetch* unit requests and receives the current PC value from the *control/status* unit (indicated by the *RequestPC* event).

2. The *instruction_fetch* unit requests and receives the desired instruction from the main memory (indicated by the *RequestI* event), obtained through the *memory_access* unit.

3. Once the *instruction_fetch* unit obtains a new instruction, it broadcasts the instruction to all other execution units through the *BroadCastI* event, making the new instruction available to them.

4. The proper unit (the *integer_arithmetic* unit in this case) decodes the instruction (indicated by the *DecodeALU* event).

5. The *integer_arithmetic* unit issues requests and obtains the necessary data through the *FetchOpr1, FetchOpr2* and *FetchCarry* events.

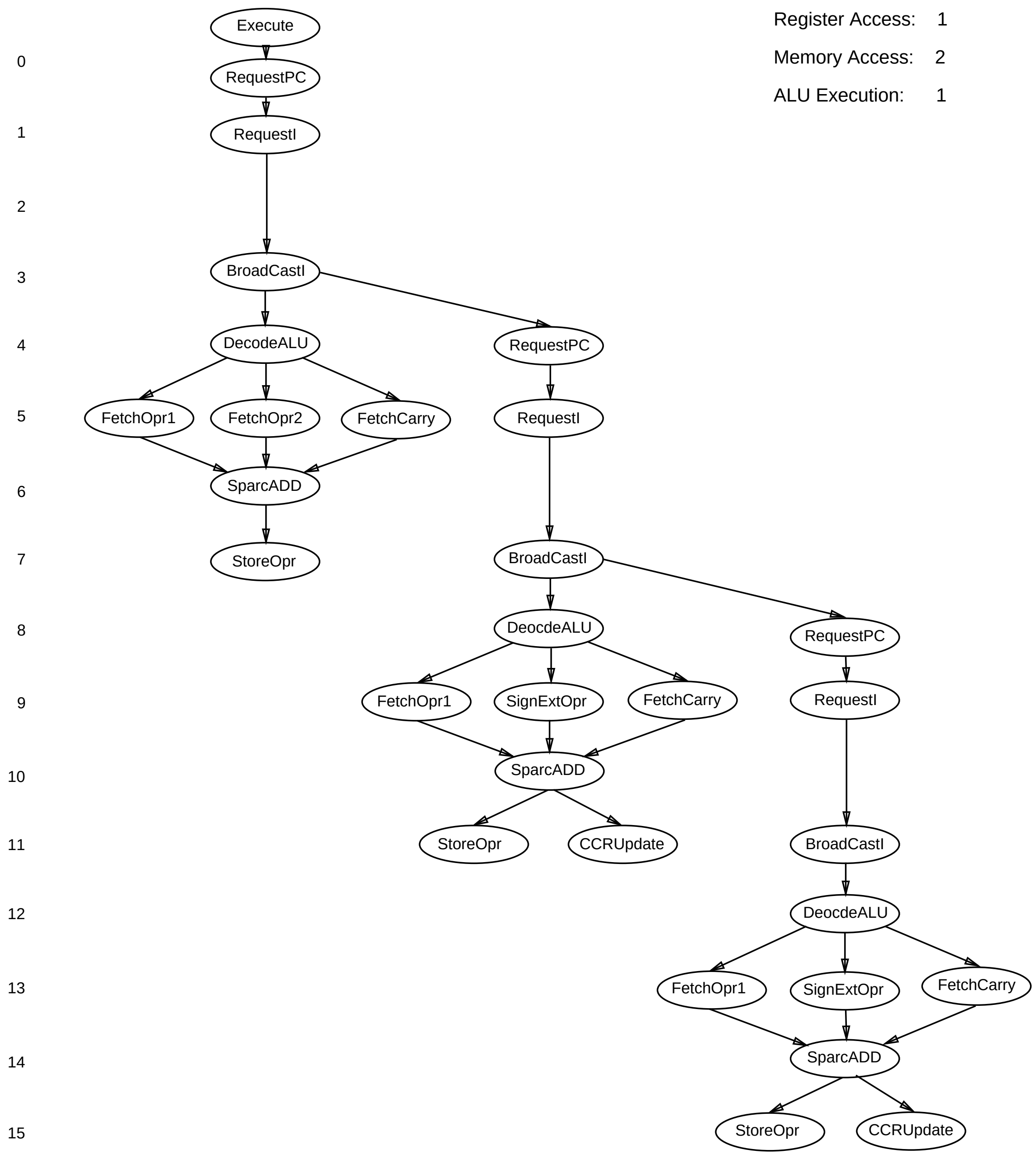

Figure 3.3: Poset of Pipelined Execution in Rapide-0.2

6. The *integer_arithmetic* unit performs the actual addition, indicated by the *SparcADD* event.

7. The *integer_arithmetic* unit stores the result back in the *integer_register set* (indicated by the *StorOpr* event).

8. The *integer_arithmetic* unit updates the contents of the condition code registers (CCR) through the *CCRUpdate* event.

The edge connecting each group is a result of the code described in the beginning of this section, with the *instruction_fetch* unit being responsible for initiating the next instruction fetch when it has finished the current one.

The time axis[4] serves to show the parallelism of the model. Since events in the same row happen at the same time, it can be clearly seen that fetching the next PC (*RequestPC*) and processing the current instruction (*DecodeALU*) happen simultaneously, as well as the data fetching (*FetchOpr1, FetchOpr2 and FetchCarry*) and the fetching of the next instruction (*RequestI*).

There is much that can be learned from the simulation, having to do with both understanding the behavior of the model, as well as analyzing simulation results through a poset. First of all, there is the same information about potential parallelism as before, indicated by all the events happening simultaneously in the figure. This same parallelism is observed both within an instruction (i.e, the data fetches mentioned in the previous paragraph), as well as between instructions (with the *instruction_fetch* unit fetching the next instruction while the ALU is processing the first one).

Another interesting point to note is that analyzing the poset is considerably simplified by how the data is presented. The human brain is good at recognizing patterns and patterns are what the result consists of. By looking at the poset one can immediately perceive which events are part of which instruction, as well as the relationship between the instructions themselves. Since the **pob** allows for cropping and filtering of events and edges, this makes the process of finding the events of interest much faster than looking through a table of numerical data or lines representing waveforms in a graphical representation of the results.

To make the figure clearer, some edges between events were deleted. The original poset contained edges between pairs of *FetchOpr1's, FetchOpr2's* (or *SignExtOpr's*), *FetchCarry's, StoreOpr* and *SparcADD* events. These edges are there because of limitation in Rapide-0.2, that makes events inside the same unit always be ordered, a problem that does not occur in Rapide-1.0. In this model, this can be looked upon as a resource contention problem, since there is only one access path for the status or integer registers. The poset representation, thus, leads one quite naturally to observe the problem of resource contention/bottleneck in the model.

The model presented in this section shows that the original architecture is flexible enough that it can be easily modified to create a pipelined implementation of the standard. This model also shows that the resulting poset contained a great amount of easily analyzable

[4] As with the example in the previous section, the time values chosen are arbitrary

data, with more information content than one finds in a linear trace, even for such simple cases.

The results presented here are enough to validate the approach and justify the investment in time and resources necessary for constructing the actual SPARC-V9 model. There are some issues related to building such a large model, though, that have to be dealt with. They will be discussed in the next chapter.

Chapter 4

Observations

Designing an executable standard involves more than just specifying the model's components and architecture. There are several related issues concerning how the model can/should be used, the procedure for testing it, the design's limitations, etc. This chapter will tackle some of these issues, describing problems that might be encountered and, whenever feasible, proposing solutions for them.

4.1 Testing

Once the executable standard is ready, the next step is testing it. Testing will have to accomplish two purposes. First, it should aid in debugging the model and assuring the design team that it behaves according to what they believe the SPARC-V9 standard states. Second, it should satisfy others, notably SPARC International, that this is indeed an executable model of the SPARC-V9 architecture, conforming to the standard. These two objectives in testing, though similar, require slightly different approaches.

When testing and debugging the model, the developers' main concern is whether the model works or not, and whether its behavior is what they expect it to be. At this point one wants to make sure that each unit is working correctly and that the generated posets show the desired implementation-independence. At this phase of the testing, it is perfectly feasible to test the model's components separately, if necessary, and integrate them to form the final model. Using simple test vectors for the model is perfectly valid at this point.

When testing to verify if the model is actually a SPARC-V9, the objectives are slightly different. It is no longer important how the model accomplishes its tasks, but only whether the final results are those that would be expected for such an architecture. At this stage one is more concerned with the final state[1] of the model than anything else.

One example might help clarify this point: when checking whether an ADD operator was implemented correctly, the designers would be interested in examining the poset and making sure that it in no way implies how the addition operation is being performed (ripple

[1]By *state* we mean the contents of the registers and the main memory

adder, carry-lookahead adder, etc.). For conformance to the standard, that is not an issue; all that is important is that the result of the addition is the expected one.

To verify that the model does indeed conform to the standard, the more straightforward approach would be to treat it as if it were a SPARC-V9 processor implemented in hardware. One could then give a copy of the model to SPARC International and let them run it through the same battery of tests to which they submit all projects they receive. This should be enough to have the model declared to be SPARC-V9 compliant.

There are, however, some problems with this approach. First, the poset generated by the model, when executing thousands of instructions, is huge. Browsing through over 100,000 events in order to identify any possible bugs is time consuming and error prone. Constraints do help in detecting and debugging errors, but it still is necessary to go over the posets manually in order to check for errors for which one forgot to write constraints.

A second problem is resource availability. There has to be enough hard disk space to store such a poset, as well as RAM and processing power for laying out the corresponding graph. A poset with 100,000 events, for example, would require about 250 Megabytes of hard disk space to store. Laying out the corresponding graph with the **pob** would take over 3 hours[2].

These problems arise due to the size of the test vector. And the reason the vector is so big is because it has to take into account pipelined, multiprocessor and superscalar implementations of the architecture. These implementations may have several incomplete instructions being processed simultaneously, all perfectly able to affect each others' outcome (generally through exceptions).

In order to simplify testing, it is important to take into account the characteristics of the model proposed in this report. The most important one is that the executable standard executes each instruction completely before starting the next one, and does not run into the problems that such a complex test vector is designed to detect. It is still a SPARC-V9 model, since the English standard states that any implementation of the architecture should behave as if it were sequential. In fact, it is the simplest of all possible SPARC-V9 implementations.

Since the conditions that lead to complex tests do not occur in this model, it might be possible to test it by running each opcode (instruction variation) through it once, or at least a small number of times. This corresponds to part of the actual SPARC test suite, more specifically the level 0 tests[Sys93]. If it can be proved that in the Rapide model the outcome of one instruction does not affect the execution of the next one (other than through changes in the state units inbetween executing instructions), then the approach should work. The resulting test vector would be smaller, and each opcode could be treated as a completely independent test, simplifying the verification of the model's correctness.

It should be pointed out that this proposed test vector does not substitute the SPARC International test suite. Rather, its use is in verifying that the generated posets are correct. Thus, when the SPARC test suite is used, there is no need to check the posets themselves, but rather one needs only to observe the final state of the registers instead.

[2] Using a SPARCStation 2 with 32 MBytes of memory.

With this in mind, the next step is determining what the test vector should be. Clearly some instructions can have only one possible outcome and thus do not need to be tested more than once. Other instructions, on the other hand, have so many possible outcomes that studying them all is infeasible. Thus, in order to come up with a thorough, but realizable test vector, a series of assumptions and guidelines are necessary. They are:

- Each opcode should be executed at least once.
- If an opcode has more than one possible decoding (for example, two register operands vs. register+immediate), all such decodings must be tested.
- If the behavior of an opcode depends on the state of one or more flags, all possible flag combinations have to be tested.
- If the behavior of an opcode may change the state of one or more flags, variations of it that change the flags (or keep them from changing) should be tested.
- If the operation works for one data value, one may assume they work for all values. For example. if it adds two numbers correctly, one can assume that it will add all numbers correctly.

These assumptions were applied to the instructions defined in appendix A of [WG94] in order to determine the test vector. Table 4.1 lists the number of instructions that have to be executed to test each opcode thoroughly. Together with tests for the exception conditions, this leads to a test that is approximately one tenth the size of the original SPARC vector.

The proposed test vector should take care of all the necessary testing for debugging and satisfying the designers as to the correctness, completeness and implementation independence of the model. It should also serve as basis for simplifying the conformance testing, in that it makes looking at the complete poset unnecessary. Instead, all that is necessary is for the user to verify the final state of the model. Tools are currently being developed to allow one to easily do this, most notably **snoop**, a register browser for SPARC-V9 log files.

4.2 Readability

The issue of readability is, possibly, one of the hardest ones to deal with, mostly because there is no absolute measure of how readable something is. Different people are used to different styles and what might be readable to one is not necessarily readable to someone else. Identifying which is the best style is not simple.

There are several things that contribute to making the code more or less readable. They range from simple topics such as naming conventions for variables and types to complex issues like the generic structure of the code for the several module generators. Finding out which is the best approach is not easy.

One way to deal with this problem is by trial and error. One should try different coding styles and show the code to as many different people as possible. This should help determine

Instruction Group	Count	Instruction Group	Count
Add	36	FMOVcc	60
BPr	12	FMOVr	12
FBfcc	30	MOVcc	176
FBPfcc	60	MOVR	24
Bicc	30	Multiply and Divide	6
BPcc	60	Multiply (32-bit)	8
Call and Link	1	Multiply Step	2
Compare and Swap	8	No Operation	1
Divide	44	Population Count	2
DONE and RETRY	2	Prefetch Data	20
FP Add and Subtract	6	Read Privileged Register	17
FP Compare	24	Read State Register	8
Convert FP to INT	6	RETURN	2
Convert FP to FP	6	SAVE and RESTORE	4
Convert INT to FP	6	SAVED and RESTORED	2
FP Move	9	SETHI	1
FP Multiply and Divide	8	Shift	24
FP Square Root	3	Software Reset	1
Flush Instruction Memory	2	Store Barrier	1
Flush Register Window	1	Store FP	10
Illegal Instruction Trap	1	Store FP in Alternate Space	6
Impl. Dependent Inst.	2	Store INT	10
Jump and Link	2	Store INT in Alternate Space	10
Load FP	10	Subtract	36
Load FP from Alt. Space	6	Swap Reg. w/ Memory	2
Load INT	16	Swap Reg. w/ Alt. Space Mem.	2
Load INT from Alt. Space	16	Tagged Add	6
LDSTUB	2	Tagged Subtract	6
LDSTUB from Alt. Space	2	Trap on INT cond. codes	116
Logical Operations	108	Write Privileged Register	30
Memory Barrier	1	Write State Register	16
Exceptions	37	**Total**	**1178**

Table 4.1: Test vector count for suggested approach

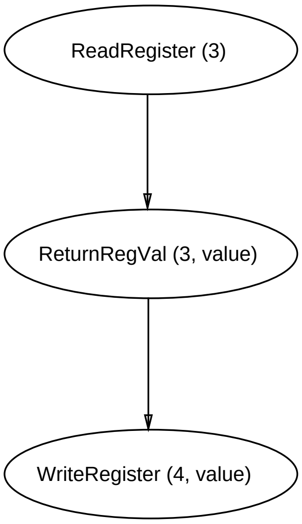

Figure 4.1: Ideal poset for the r[4] := r[3] operation

which styles are, if not better, at least more widespread and thus easier to understand by a larger part of the community. Appendix A presents a set of coding guidelines that have been defined for the coding of this project. This is a tentative series of guidelines that will probably have to be refined as the model is built.

So far, we have discussed what can be done to improve the readability of the source code, for when it is to be used as a reference source. There is also the issue of how to make the generated poset itself readable. This problem is more complex than it seems, as will be shown in the following example.

Suppose one wants to write code for assigning the values stored in the integer register r3 to integer register r4. The first suggestion is to write the following piece of code for the assignment:

r[4] := r[3];

Supposing that r is an object of the *integer register set* type and that [ ] is the function for accessing the respective register, this code is quite readable, and its meaning in unambiguous (a copy of the contents of register 3 is stored in register 4). This statement, though simple, does not generate any events that would appear in a poset[3]. This makes it necessary to add more statements to the source code, in order to make the corresponding poset useful.

An interesting poset for the operation described above is shown in figure 4.1. In this poset we see that the act of assigning the value of r[3] to r[4] consists of three parts: asking for the desired register value (*ReadRegister*), retrieving the value of register r[3] (*ReturnRegVal*), and then writing the result to register r[4] (*WriteReg*). The corresponding code is:

[3]The events that appear on a poset are events corresponding to actions defined in the language. Function calls generate corresponding *call* and *return* events, but these are not seen in the final poset

```
ReadRegister(3);
await (?val in SPARCint_t)
      ReturnRegVal(3,?val) =>
      WriteRegister(4,?val);
```

ReadRegister(3) is an event sent to the Integer Register Set requesting the contents of register r[3]. The code then specifies that one should wait until getting the *ReturnRegVal* event, which contains the requested value. This value is then assigned to the new register through the *WriteRegister* event. This code generates the poset in figure 4.1 and causes the same state change as the previous piece of code. But, as can be seen, it is much more complex and harder to understand.

This, then, is the main problem with readability. The coding technique that makes the source code readable does not necessarily contribute to making posets readable, and vice-versa. A compromise must be achieved, so that the resulting source code and poset are, if not ideal, at least understandable.

4.3 Constraints

A constraint is a rule or condition that checks, restricts, or compels a system to avoid or perform some specific action. In this paper's context, the emphasis is on the word *checks*. A constraint is, thus, a rule that states how a system should or should not behave. It does not force the desired behavior, but instead defines the space of allowed behaviors over which an implementation may vary.

Constraints can be used to completely specify and build a totally new class of models known as *constraint-based* models. Unlike executable simulation models, such as the one described in this report, constraint-based models do not show how an implementation is expected to behave, but just list all the rules that the implementation must satisfy if it is to be considered valid. For this to be possible, the constraints must be sufficient to correctly and completely specify the desired behavior. It is also helpful if the constraints could be used to automate the verification of an implementation's correctness. As was mentioned before, Rapide has all the necessary characteristics for making this happen.

In the context of hardware modeling, it is interesting to come up with a taxonomy for constraints, categorizing them according to what aspect of the architecture they refer to. Instruction set architectures are usually specified by stating the registers and functional units present in the model, the expected behavior whenever certain inputs occur, and conditions that are recognized as error conditions. This suggests three different classes of constraints:

- **Structural constraints:** Specify the structural attributes of the constituents of the model, indicating their existence and providing information about their size. This includes things like the data width, the size of the register file and the size of the address space.

- **Behavior constraints:** Specify the expected or desired behavior of some part of the model, such as, for example, how a processor should react to a division by zero exception.

- **Error constraints:** Are used to specify behaviors which should never be observed in a correct implementation. An example of this would be a sum operation that yields a wrong result.

There are several ways to represent such constraints in a form that can be processed by a computer. Structural constraints, for example, can be represented as variables with limits on the values they can store. This is already found in many programming languages. C, for example, allows one to define a variable to be of type **enum**, listing all the possible values that the specified variable may store[KR88]. Error and behavior constraints can be expressed in several different ways including temporal logic[Fuj87] and finite state machines[BEP93].

Through the use of an adequate programming/specification language one can use constraints such as the ones described above to build a model of a system. This model can find many uses, but the most important one is in design verification. An actual implementation can be built and its simulation results can then be checked against the constraint-based model for compliance. If the constraint-based model is correct and complete, conformance to it guarantees that the implementation being verified is also correct. Note that this model is not a tool for formal verification, i.e., it does not check the state space of an implementation and guarantees its correctness. Instead, it speeds up the design process by guaranteeing that simulation results are valid and do not violate the standard.

There is a straightforward relationship in Rapide between interfaces and constraints. The several parts of an interface correspond directly to specific types of constraints:

- **Declaration part:** This part is used to specify all the components of the interface that are provided or required by an object of that type so that it can communicate with other objects. Thus, it describes the structural constraint specifying which resources have to be available (such as register), and/or mechanisms for accessing these resources.

- **Behavior part:** This part describes a set of reactive programming rules, that indicate how an object of that type should behave, which are nothing more than the behavior constraints defined earlier. One important observation is that a module generator should be used to code the actual behavior of an object of the type. If no generator is defined, the behavior part will be used to automatically generate one; otherwise it will be used as a constraint.

- **Constraint part:** This part is used to define patterns that must always (or never) occur in a Rapide simulation of an object of that type. It is used for the error constraints, which indicate patterns that one never wants to find in a simulation.

The structure and architecture of a constraint-based model of the SPARC-V9 would be very much like the one described previously in this report. Most of what would change is that what was before written as the behavior of a module now becomes a constraint as to how it should behave. Since the original model is implementation-independent, this characteristic should be maintained, resulting in a model that can be used as an automatic verification tool for actual designs.

What remains to be seen is how powerful Rapide's constraint language is, and how it can be used to efficiently describe a constraint-based model of the SPARC-V9. Each kind of constraint has intrinsic characteristics that may make them easier or harder to code efficiently. The following subsections will show examples of each kind of constraint, and determine how easily they can be implemented in Rapide.

4.3.1 Example 1: Error Constraint

The SPARC-V9 standard specifies some limits on the addresses for accessing data stored in memory. This requirement is a good example for an error constraint, since it specifies conditions that one never wants to see. To be more precise:

- There are *no* restriction on the addresses of byte accesses.
- Halfword (16-bit) accesses must be aligned on 2-byte boundaries.
- Word (32-bit) accesses must be aligned on 4-byte boundaries.
- Doubleword (64-bit) accesses must be aligned on 8-byte boundaries.
- Quadword (128-bit) accesses must be aligned on 16-byte boundaries.

This condition can be expressed as four error constraints specified in the memory access unit, stating that one does not want to see unaligned memory accesses. In Rapide, this takes the following form:

```
never(?a in SPARCint_t, ?m in AccessMode)
        ReadReq(?a, ?m) where (?m = HALFWORD and (?a mod 2) /= 0);
never(?a in SPARCint_t, ?m in AccessMode)
        ReadReq(?a, ?m) where (?m = WORD and (?a mod 4) /= 0);
never(?a in SPARCint_t, ?m in AccessMode)
        ReadReq(?a, ?m) where (?m = DOUBLEWORD and (?a mod 8) /= 0);
never(?a in SPARCint_t, ?m in AccessMode)
        ReadReq(?a, ?m) where (?m = QUADWORD and (?a  mod 16) /= 0);
```

These constraints are all stating that, in a poset, there should never be a read request to memory of the proper kind (HALFWORD, WORD, DOUBLEWORD and QUADWORD) with a non-valid memory address. There are no constraints for byte accesses because there is no alignment condition for that kind of access.

Figure 4.2 shows the poset resulting from a simulation that violates these constraints. Notice the presence of the *inconsistent* event, indicating that a constraint has been violated.

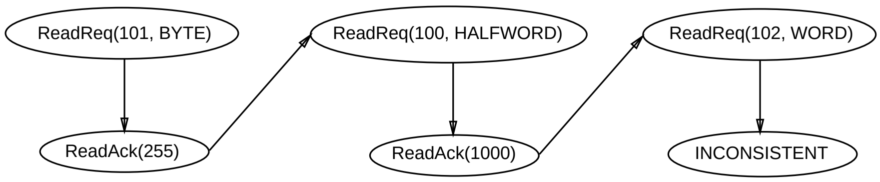

Figure 4.2: Example of an error constraint violation

In this case, the violation happened because the third *ReadRequest* event had a word address (address 102) that was not properly aligned (it is not a multiple of four).

This kind of error is detected automatically by the simulation, and the tools can easily identify it. The causality edges in the poset can then allow one to trace the *inconsistent* event back to its cause, determining the sequence that generated the error.

4.3.2 Example 2: Behavior Constraint

The SPARC-V9 manual consists mostly of behavior constraints, specifying what an instruction should do, and which state changes have to be made. The branch instructions are an example of this. They all differ in the exact condition they check, but they share one common template: they all evaluate a condition, calculate the effective address and then possibly update the nPC register with that new value when the branch takes place. This condition is written in Rapide in the following way:

match(Decode →
(Condition_Evaluation ~ Effective_Address_Calculation) →
(Update_nPC **or Empty**) →
Executed)^(~ *);

Decode and *Executed* are the events that mark the boundary of execution of any specific opcode in the model. What the constraint above says is that, if the instruction is identified as a branch instruction, it should be followed by an evaluation of the condition (the *Condition_Evaluation* event) and a calculation of the effective address (the *Effective_Address_Calculation* event) in *no* particular order (that's what the '~' binary pattern operator means) and that should result in the nPC register being updated or not. This particular constraint does not check if the branch should have been taken or not depending on the condition being evaluated (there should be other constraints for that). Instead, it just specifies the general form of the behavior of such opcodes.

Figure 4.3 shows an example of this constraint being violated. In the poset, one sees that the implementation erroneously wrote the effective address to the PC register instead of the nPC register, and that is a pattern not expected by the constraint.

Just as in the case of error constraints, this constraint can be used to automatically check both complete implementations and components being developed. Straightforward

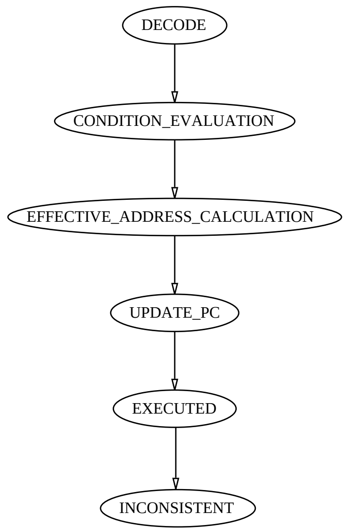

Figure 4.3: Example of behavior constraint violation

mapping could be applied to the complete implementation, while module generators could be plugged into the model when one just wants to test components.

4.3.3 Example 3: Structural Constraint

Structural constraints are a little bit harder to check. There are two ways a structure can violate the specification: either it is not present when it should be, or it has the wrong format(wrong size, for example). Unexpected structures that are present are not a problem, since having something extra does not invalidate an architecture (unless specifically prohibited by the standard, which is not the case here). Unfortunately, these are *static* characteristics and posets show the *dynamic* behavior of a model so there is no way for the structural constraints to actually generate *inconsistent* events when a simulation is run.

Structural constraints should, therefore, be checked during the phase when the model is being compiled, since at this point the compiler can check the structure of the suggested implementation and see if it conforms to that which is defined in the interface. Take, for example, the structural constraint expressing the need for the presence of both a program counter (PC) and a next program counter (nPC) in a SPARC-V9 implementation. Both should be 64-bit registers that can be both read from and written to. This is how one could express such a constraint in Rapide, in the declaration part of an interface:

PC : ReadWriteReg_t **is** ReadWriteReg_g(64); -- 64-bit wide
nPC : ReadWriteReg_t **is** ReadWriteReg_g(64); -- 64-bit wide

ReadWriteReg_t is one of the basic types presented in section 2.3 that specify the structure and behavior of 64-bit registers. Any implementation of the model should have these registers present. Compilation of a module which does not have these components, or has the wrong structures associated to them would generate compilation errors, thus indicating a violation of the constraint.

This approach is, unfortunately, limited to modules that are being designed using the executable model described in this paper as their basis. They would also have to use the basic types defined in 2.3. That is, if one is following this general architecture and creating separate units for the control/status registers, floating point registers and integer registers, checking structural constraints could be performed automatically. Fortunately, the only units that would have to conform to this format are the state units (see section 2.4) since it is there that all structural constraints are located.

If an implementation chose not to follow this organization for the several modules, there would be no way to easily verify the model's structural correctness. An approach to solving this problem would consist of having a tool for comparing the source code of the implementation and the model, extract the relevant structures and see if they match or not (graph-isomorphism algorithms come to mind here). This would probably require the use of annotation languages, along the lines of ANNA[KBL80] and VAL/VHDL[ALG+90] in order to provide a formal specification of the desired structures. A tool for this has not yet been written.

This section presented three kinds of constraints: error, behavior and structural. Rapide has mechanisms for checking the first two automatically, with no more inconvenience to the designer than defining maps from his implementation to the constraint-based model. Checking structural constraints is more complicated, since straightforward mapping does not work here. Still, tools can be developed to perform this task.

4.4 Mapping

In section 1.2 we presented mapping as a mechanism for translating a set of events in one domain into events in another. In Rapide this is done by describing event patterns in one domain and binding them to patterns in the second. These patterns can be arbitrarily complex, allowing one to do something as simple as mapping an event to another event, or as complex as mapping entire computations to a single event (and vice-versa). Mapping is a very flexible and powerful concept.

There are many uses for mapping, but in the context of modeling instruction set architectures and standards, there are two applications which are of interest to us. First, mapping can be used to allow one to use already existing modules in the design without having to modify them at all. Second, it allows one to check complete designs against the standard automatically, thus verifying their conformance to the specification.

Using mapping to connect pre-constructed modules accelerates the design process. Since the module has already been written and debugged, no time is lost in rediscovering algorithms or verifying if they have been implemented correctly. By using maps, it is possible to get around idiosyncrasies in the code that would force one to change one's model in order to be able to use the package. For example, if one has a floating point package available that requires sequential events for loading the data into the floating point unit, but the model issues independent events for loading the data, mapping would take care of changing the independent requests into a fully ordered one in one simple step.

The second use of mapping is, as was mentioned above, in verifying the correctness of a complete model. Suppose, for example, that one builds a pipeline model of the SPARC-V9 architecture using Rapide and wants to check it against the standard (either the executable one, or the constraint-based model suggested in the previous section). Because of its implementation-specific nature, there might be events in the pipeline module that do not correspond exactly to the standard, or vice-versa. For example, it is common in pipeline architectures to forward a value to other pipeline stages in order to diminish the number of stalls in the pipeline[HP90]. This event does not appear in a sequential model and, therefore, mapping would be necessary to mask it when doing the verification against the standard. This feature makes mapping a useful mechanism in design verification.

The same features that make mapping a powerful mechanism also make it a problem. Namely, one has the freedom to map any event pattern to another, without any restrictions. This enables one to make the model of a clock, for example, look like a car or even a distributed database. Through mappings, it is very easy to inadvertently mask or introduce errors in a computation.

The authors do not yet have much experience with mapping in order to provide an effective solution to this problem. Techniques, guidelines and possibly software tools have to be developed to help designers in constructing correct mappings from one domain to another. Until this is done, careful coding and painstaking verification are the only way to guarantee the correctness of results when mapping is used.

Mapping makes it easy to verify the correctness of an implementation by defining rules that map it to the standard and then checking for constraint violations. The same features that enable one to do this, though, also allow one to introduce or mask errors in a model unless careful attention is paid to the mapping process. Mapping, though a powerful mechanism for design verification, should be used with care until a better understanding of the process is attained.

Chapter 5

Conclusion

This report described an approach to building an implementation-independent model of the SPARC-V9 instruction set architecture. This model was designed to be used as an executable reference tool, helping designers in understanding the intricacies of the architecture.

In order to be an executable standard, the model must have several important characteristics. It has to be *complete* and *correct* in order to be a valid reference source; it has to be *readable* in order to be accessible to designers; it has to be *executable* so as to provide on-line help in understanding the model; it has to be *precise* so that there would be no misunderstanding as to what each aspect of the architecture did; it has to be *implementation-independent* so as not to add any conditions to the architecture which were not specified in the English language manual. Rapide-1.0 was chosen as the language for implementing this model because it has all the necessary features to satisfy these objectives.

The architecture proposed consisted of defining basic building blocks for constructing the model, as well as grouping opcodes according to their functionality into components. These components were then connected through loosely coupled buses and a communication protocol was defined, creating the model's architecture. There were a total of eight basic building blocks, twelve components and six buses.

A simple model was built, using Rapide-0.2, to test the validity of the proposed architecture. The authors were able to get the model up and running in a very short time, as well as modifying it to create a pipelined version of the model. This satisfied the authors as to the feasibility of the approach.

The proposed approach does not deal with all the issues that might arise in the process of creating the actual model. These various issues include: the definition of a testing procedure to verify the correctness and adequacy of the module; the definition of guidelines for ensuring the readability of both the source code for the model and the posets generated during a simulation run; the specification of constraints for the model's behavior and structure; and the definition of guidelines for creating maps from actual implementations for comparison against the executable standard. Partial solutions have been presented for some of these issues, but others still need further exploration.

The authors have started working on the actual implementation of the model described in this report. It is estimated that this work will take about one man-year to complete and that the model will have approximately 15,000 lines of code. Once the model is complete a considerable amount of time will still have to be spent on developing testing procedures, analyzing posets and redesigning the model so as to better satisfy the initial goals.

This project, though complex enough in itself, also opens the doors for several different areas of research. Some topics that might possibly be explored in the future in more detail are:

- **Interconnection with other simulation languages:** Though Rapide is a very powerful and useful prototyping languages, it is very high-level. It might be interesting, as models develop, to use other languages more suited for specific domains (for example, using Verilog or VHDL for a more precise description of a hardware component of a model). In this case, it would be interesting to allow for the seamless integration of something described in such a language with the original Rapide-1.0 high-level model.

- **Exploration of different architectures:** This report has described an implementation-independent model of the SPARC-V9 instruction set architecture. It would be interesting to use this model as a basis for implementation-dependent models, that could then be used for performance/resource analysis. Some interesting models to build would be superscalar models, pipeline models and a resource-exploration model that would take advantage of Rapide's dynamic object creation facility to create resources such as ALUs and register files as they were needed. This model would be quite useful in evaluating the maximum number of each type of resource needed for a specific program.

- **Poset analysis of hardware models:** Posets, with the added causality information, add a new aspect to the hardware simulation result. Algorithms have to be developed to help understand what this extra information means. For example, one should be able to identify which edges in a simulation result happen due to resource limitations and which ones are due to program dependencies.

- **Posets and linear traces:** Posets and linear traces are two different views of a simulation result. To be more precise, a poset represents a set of possible linear traces. There are many mathematical explorations and algorithms to be devised to answer questions such as how many different linear traces a specific poset represents, whether a linear trace is contained in a given poset or not, generating all the linear traces corresponding to a poset, or even generating a poset that satisfies a given set of linear traces.

- **Poset presentation:** Posets with more than a few dozen events in them are not easily presented on a computer screen. Methods and tools have to be developed to allow people to easily scan huge posets and extract the information they desire from a poset. Also, it would be interesting to have tools that would guide the user and help one zero in on the areas of interest.

The project described in this report will be useful in several different ways. First, it will provide a tool for helping designers understand the SPARC-V9 instruction set architecture standard. Second, it will serve as a test case for the Rapide-1.0 toolset. Finally, it will serve as the basis for much future work in exploring Rapide, posets and the SPARC-V9 architecture. The authors are proud to be part of this effort.

Acknowledgements

The authors would like to thank several people for their help in making this work possible. David Weaver, from Sun Microsystems, was very helpful and patient in answering our questions and making sure we had updated information on the SPARC-V9 standard. Ernest Carlson Jr., from SPARC International, was very helpful in getting us information on the testing procedure. Finally, the Program Analysis and Verification Group of the Computer Systems Laboratory at Stanford University, including Doug Bryan, Larry Augustin, James Vera, Walter Mann and John Kenney was especially helpful in answering all our Rapide questions and providing us with working compilers whenever we needed them.

Bibliography

[ALG^{+}90] Larry M. Augustin, David C. Luckham, Benoit A. Gennart, Youm Huh, and Alec G. Stanculescu. *Hardware Design and Simulation in VAL/VHDL*. Kluwer Academic Publishers, October 1990. 322 pages.

[Bar81] M. R. Barbacci. Instruction set processor specification(ISPS): The notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[BEP93] Dominique Borrione, Hans Eveking, and Laurence Pierre. Formal proofs from hdl descriptions. In Jean P. Mermet, editor, *Fundamentals and Standards in Hardware Description Languages*, pages 155–193. Kluwer Academic Publishers, 1993.

[Bry92] Doug Bryan. Rapide–0.2 language and tool-set overview. Technical Note CSL–TN–92–387, Computer Systems Lab, Stanford University, February 1992.

[Fid91] Colin J. Fidge. Logical time in distributed systems. *Computer*, 24(8):28–33, August 1991.

[Fuj87] Masahiro Fujita. Hardware verification based on hdl sources. In R. W. Hartenstein, editor, *Hardware Description Languages*, pages 283–312. North-Holland, 1987.

[GT88] Daniel D. Gajski and Donald E. Thomas. Introduction to silicon compilation. In Daniel D. Gajski, editor, *Silicon Compilation*, chapter 1, pages 1–48. Addison-Wesley, 1988.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[IEE85] IEEE. IEEE standard 754-1985 for binary floating point arithmetic, 1985.

[Int92] SPARC International. *The SPARC Architecture Manual - Version 8*. PTR Prentice Hall, 1992.

[KBL80] B. Krieg-Brückner and D. C. Luckham. Anna: Towards a language for annotating Ada programs. *ACM SIGPLAN Notices*, 15(11):128–138, 1980.

[KR88] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[LVM95] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical Report CSL–TR–95–674, Computer Systems Lab, Stanford University, July 1995.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms.* Elsevier Science Publishers, 1988. Also in: Report No. SFB124P38/88, Dept. of Computer Science, University of Kaiserslautern.

[Sys93] HaL Computer Systems. *AVPGEN User's Guide.* HaL Computer Systems, 1315 Dell Avenue, Campbell CA 95008, final draft edition, April 1993. Final Draft.

[Tea94a] Rapide Design Team. *The Rapide-1 Architectures Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94b] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94c] Rapide Design Team. *The Rapide-1 Specification Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94d] Rapide Design Team. *The Rapide-1 Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[TM91] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[VHD87] IEEE, Inc., 345 East 47th Street, New York, NY, 10017. *IEEE Standard VHDL Language Reference Manual*, March 1987. IEEE Standard 1076–1987.

[WG94] David L. Weaver and Tom Germond. *The SPARC Architecture Manual - Version 9.* PTR Prentice Hall, November 1994.

Appendix A

Coding Guidelines

A.1 Observations

The model of the SPARC-V9 processor described in this paper is intended to be used as an executable standard. As such, one of its necessary characteristics is readability. By that we mean that the model should be clear; a reader should be able to understand the model easily by looking at the code, and use that to answer questions. To make the code readable, a series of guidelines have to be established, ensuring coherence and uniformity. If these guidelines are followed, the resulting code should be both understandable and correct.

The guidelines presented here attempt to make the model readable at the source level by dealing with several distinct aspects of the code. First, they try to define a coding style, ensuring that the whole code has a consistent look-and-feel. Second, they present some templates (or ready-made solutions) for some parts of the design, which guarantee a correct and uniform implementation of a particular part. Finally, these guidelines provide some information about what should (or should not) be in the model.

These guidelines are not complete. For instance, they do not deal with the topic of constraints, exceptions, or poset readability. It is expected, as the project develops and style issues are brought forth, that these guidelines might change. Also, one should remember that these are not cast-iron rules, but just ideas on how the actual coding should proceed. Here, like in many other places, common sense should prevail.

A.2 Guidelines

The following are the current guidelines governing the coding of the model:

1. *All type interface names should end with '_t'.* The objective of this is to make sure that the reader knows at a glance whether what he is reading is the description of a type or an instance of an object of that type.

2. *All module generator names should end with '_g'.* The objective here is the same as in the previous item; uniformity in naming parts so as to simplify understanding of the code.

3. *Avoid elegant, but hard to understand algorithms.* The objective here is having readable code, not clever solutions that are hard to understand. It does not matter if the clever solution needs only one line of source code while the readable one needs five, as long as the latter is easier to understand.

4. *Type interface descriptions should not have a behavior part.* Trying to put too much information in one part is a guaranteed way to make the code complex and hard to read. By eliminating the behavior from the type interface and putting it in the module generator, one diminishes this complexity and makes the model easier to understand. The interface should provide information about what a type has to offer and what it needs, with the constraints giving an idea of what the poset behavior should conform to. If the reader feels the need to understand how some activity is actually executed, he can look at the code of the corresponding module generator or the poset generated by executing the model.

5. *State units should have as little behavior associated to them as possible.* State units, as the name implies, are intended to model storage, not behavior implementation. Thus, any behavior associated with them should be limited to either storing or retrieving information stored in the unit. There are, of course, some cases where behavior has to be added, such as is the case in the *integer register set*, in which a function is necessary to map the requested register index to the actual register in the register file.

6. *The code for each execution unit (with the exception of the Instruction Fetch and Memory Access units) should contain the following template in the beginning of its behavior:*

   Trigger(?i) **where** valid(?i) => decode(?i); Issue(?i);;

   Trigger is an *in* (public) action that should have as a parameter the instruction that might possibly be executed by the unit. It is intended to be connected to the Broadcast action generated by the Instruction Fetch Unit.

   valid is an internal function returning a boolean. It is used to indicate if the instruction it receives as a parameter is to be executed by this unit or not.

   decode is the function that actually decodes the instruction and sets the values of several internal state variables such as rs1, rs2, etc.

   Issue is the internal action which actually starts the internal execution of the instruction. It should be used as the pattern for several rules of the form '**Issue**() where *condition-for-instruction* =>'

   This structure has two important advantages. First, it gives a uniform look to the execution units, allowing the reader to go directly to the execution of the instructions themselves, without worrying about how they are decoded. The second advantage is that this kind of organization allows one to create a one-to-one correspondence between each **Issue** trigger and body with an opcode, thus simplifying the cross-referencing process between the English standard and the source code.

7. *Whenever possible, mention the section of the English standard that the code refers to, whether it is the definition of an object (i.e. register) or the description of a behavior.* The objective of this is to provide a mapping from the code to the manual, thus making it easy for someone to verify the meaning of the code (if necessary). This should also aid when debugging the code for correct behavior.

8. *Keep event generation to the minimum necessary. Only events that are part of the poset that one wants to produce, or those that are necessary in order to create the desired causality relations, should be used.* Events are what show up on posets and the more of them there are, the more complex the poset will be. Thus, less events mean greater readability.

These guidelines are sufficient as a starting point for constructing the model's source code. As the model begins to take shape, other guidelines might be added to the ones already here, and some of them might be modified or even abandoned. This is one issue that will only be fully resolved when the model is fully implemented.

Appendix B

Rapide Interface for Basic Types

This appendix contains the formal definition of the basic types, written in Rapide. Each section contains the interface definition for one of the basic types.

B.1 ReadReg_t

```
-- READREG_T : read only register
--
type ReadReg_t is interface

    [ ]       : function (index : integer) return ReadReg_t;
    field     : function (hi, lo : integer) return ReadReg_t;
    regsize   : function () return integer;

    inval     : function () return integer;
    sform     : function () return string;

end ReadReg_t;
```

B.2 ReadWriteReg_t

```
-- READWRITEREG_T : read and write register
--
type ReadWriteReg_t is interface

    -- access functions
    [ ]       : function (index : integer) return ref(ReadWriteReg_t);
```

```
    field      : function (hi, lo : integer) return ref(ReadWriteReg_t);

    -- arithmetic functions
    ~          : function () return ReadWriteReg_t;
    ##         : function (tail : ReadWriteReg_t) return ReadWriteReg_t;

    -- attribute functions
    :=         : function (value : integer);
    :=         : function (value : ReadWriteReg_t);
    :=         : function (value : string);

    -- miscellaneous
    regsize    : function () return integer;
    intval     : function () return integer;
    sform      : function () return string;

end ReadWriteReg_t;
```

B.3 UpCounter_t

```
-- UPCOUNTER_T : upwards counter
--
type UpCounter_t is interface

    include ReadWriteReg_t;

    reset : function ();
    inc   : function () return ReadReg_t;

end UpCounter_t;
```

B.4 UpDownCounter_t

```
-- UPDOWNCOUNTER_T : bidirectional counter
--
type UpDownCounter_t is interface

    include UpCounter_t;

    dec : function () return ReadReg_t;
```

```
end UpDownCounter_t;
```

B.5 DispCounter_t

```
-- DISPCOUNTER_T : displacement counter
--
type DispCounter_t is interface

    include ReadWriteReg_t;

    -- arithmetic functions
    +    : function (value : ReadReg_t) return ReadReg_t;
    +    : function (value : integer) return ReadReg_t;

end DispCounter_t;
```

B.6 SPARCint_t

```
-- SPARCint_t : integer register type
--
type SPARCint_t is interface

    include ReadWriteReg_t;

    -- arithmetic and logical operations
    ~      : function () return SPARCint_t;
    neg    : function () return SPARCint_t;

    &      : function (value : SPARCint_t) return SPARCint_t;
    or     : function (value : SPARCint_t) return SPARCint_t;
    xor    : function (value : SPARCint_t) return SPARCint_t;
    +      : function (value : SPARCint_t) return SPARCint_t;
    -      : function (value : SPARCint_t) return SPARCint_t;
    *      : function (value : SPARCint_t) return SPARCint_t;
    /      : function (value : SPARCint_t) return SPARCint_t;
    mod    : function (value : SPARCint_t) return SPARCint_t;

    &      : function (value : integer) return SPARCint_t;
    or     : function (value : integer) return SPARCint_t;
```

```
    xor     : function (value : integer) return SPARCint_t;
    +       : function (value : integer) return SPARCint_t;
    −       : function (value : integer) return SPARCint_t;
    ∗       : function (value : integer) return SPARCint_t;
    /       : function (value : integer) return SPARCint_t;
    mod     : function (value : integer) return SPARCint_t;

    -- shift operators
    shl     : function (shift_count: integer) return SPARCint_t;
    shr     : function (shift_count: integer) return SPARCint_t;
    ashiftr : function (shift_count: integer) return SPARCint_t;

    -- comparison operators
    =       : function (value : integer) return boolean;
    <       : function (value : integer) return boolean;
    >       : function (value : integer) return boolean;
    >=      : function (value : integer) return boolean;
    <=      : function (value : integer) return boolean;
    /=      : function (value : integer) return boolean;

    =       : function (value : SPARCint_t) return boolean;
    <       : function (value : SPARCint_t) return boolean;
    >       : function (value : SPARCint_t) return boolean;
    >=      : function (value : SPARCint_t) return boolean;
    <=      : function (value : SPARCint_t) return boolean;
    /=      : function (value : SPARCint_t) return boolean;

    -- attribute functions
    :=      : function (value : SPARCint_t);

end SPARCint_t;
```

B.7 SPARCreal_t

```
-- SPARCreal_t : floating point register type
--
type SPARCreal_t is interface

    include ReadWriteReg_t;

    +        : function (value : SPARCreal_t) return SPARCreal_t;
    −        : function (value : SPARCreal_t) return SPARCreal_t;
```

```
    *        : function (value : SPARCreal_t) return SPARCreal_t;
    /        : function (value : SPARCreal_t) return SPARCreal_t;
    sqrt     : function () return SPARCreal_t;

    -- comparison operators
    =        : function (value : SPARCreal_t) return boolean;
    <        : function (value : SPARCreal_t) return boolean;
    >        : function (value : SPARCreal_t) return boolean;
    >=       : function (value : SPARCreal_t) return boolean;
    <=       : function (value : SPARCreal_t) return boolean;
    /=       : function (value : SPARCreal_t) return boolean;

    -- attribute functions
    :=       : function (value : SPARCreal_t);

end SPARCreal_t;
```

B.8 RegStack_t

```
-- REGSTACK_T : register stack type
--
type RegStack_t is interface

    include ReadWriteReg_t;

    pop : function () return ReadReg_t;
    top : function () return ReadReg_t;

end RegStack_t;
```