

# Application Specific Electronic Design Synthesis

William R. Bush

Phase I Report, July 1994

SBIR Topic Number: AF93 - 117

## TECHNICAL ABSTRACT:

The Air Force and other DoD organizations are continuously developing new electronic components for their weapons and other systems. In this effort they are in need of tools to aid in the automated design, or synthesis, of these components. One component of particular importance is the computer processor.

The research performed here applies a new processor synthesis technique, based on the methods of RISC computer design, to the synthesis of a widely used, commercially available microprocessor of interest to the Air Force, the SPARC. This technique automatically generates, from a single instruction set architecture specification, different implementations optimized for different applications, with the different implementations based on the instruction frequencies of the applications. Various implementations of the SPARC have been generated.

During Phase I, proof-of-concept tests of the synthesis technique were performed, and the strengths and weaknesses of the synthesis system were evaluated. Phase II will involve the construction and demonstration of a robust tool based on the current prototype. Furthermore, reference manuals and user guides will be developed and training provided to USAF personnel in its use.

## ANTICIPATED BENEFITS / POTENTIAL COMMERCIAL APPLICATIONS:

The synthesis tool used in this research automates the design of computer processors and, potentially, other electronic components. It speeds up the process of processor design by rapidly producing an optimized prototype implementation. It also improves the quality of the results by automatically specializing computer implementations for particular applications. Its possible uses extend to both integrated circuits and electronic systems constructed from standard parts. The synthesis technique is new, and has not been commercialized. During Phase III it will be; testing, marketing, support, and maintenance plans will be developed.

## KEY WORDS:

APPLICATION SPECIFIC ELECTRONIC DESIGN SYNTHESIS,  
AUTOMATED RISC-STYLE DESIGN, COMPUTER AIDED DESIGN (CAD),  
HARDWARE SPECIFICATION, HIGH-LEVEL HARDWARE SYNTHESIS,  
MICROPROCESSOR SYNTHESIS, SPARC, VHDL

## TABLE OF CONTENTS

1.0	Executive Summary
2.0	Problem Definition and Significance of the Innovations
3.0	Phase I Technical Objectives
4.0	Technical Approach
5.0	Technical Implementation
6.0	Research Results
7.0	Conclusions and Recommendation
8.0	Bibliography
9.0	Appendix: The Prolog SPARC Specification
10.0	Appendix: Improving the Specification
11.0	Appendix: Translating Prolog into VHDL

## 1. EXECUTIVE SUMMARY

Traditional, human-intensive methods of electronic hardware design are relatively slow and somewhat error prone. This has led to the growth of the electronic computer aided design industry, and the proliferation of computer aided design (CAD) tools. Integrated circuits, in particular, have slow and expensive design iterations (due to mask generation and chip fabrication), and are hard to debug. Additionally, hardware designers have lacked the development tools possessed by software designers, who supported their own activities with tools such as compilers, debuggers, libraries, and software engineering environments.

The work performed here takes advantage of recent research involving the high level specification and automatic generation of hardware. A high level optimization tool, called Viper, was evaluated. It rapidly produces application specific hardware, microprocessors in particular.

The tool uses a new optimization technique, based on the methods of RISC computer design. It automatically synthesizes, from a single instruction set architecture specification, different implementations optimized for different applications, with the different implementations based on the instruction frequencies of the applications. The operation of the tool is based on three principles [Viper]:

- Given constraints on how much hardware can be generated, process instructions in the input specification in order of importance, allocating hardware for the most important operations first.
- Do not allocate hardware for unused instructions.
- Transform the specification using a well known method for microcode compilation called trace scheduling ([Trace]) that can increase the speed of execution of common instructions by increasing the potential parallelism in such instructions.

To evaluate the tool, the following major tasks were performed:

- A specification of the commercially available SPARC RISC microprocessor was written in a format suitable for input to the synthesis tool.
- Various implementations of the SPARC were generated, using the high level synthesis optimization techniques of the tool to explore the design space.

Additionally, a VHDL SPARC specification was written to explore the feasibility of using VHDL as a hardware input specification language to the tool, and various issues relating to the implementation of the tool were examined.

It was found that:

- Once a SPARC processor specification in the proper format had been written, implementations were in general generated quickly and relatively easily.
- Specialized microprocessors were essentially trivial to produce.
- At the cost of more hardware, optimized designs (40% faster in terms of throughput) were easily generated.
- The tool was robust enough to handle the SPARC design, but it pushed the limits of the implementation. Any further development will require a reimplementaion, preferably in C++.
- Using VHDL for specification was substantially easier and more natural than using the tool's original input language. VHDL's rich control structures made specifications much more readable, and the ability to specify hardware structure made it much easier to define a library of functional units.
- The tool's user interface and feedback to the user could be greatly improved.

## 2. PROBLEM DEFINITION AND SIGNIFICANCE OF THE INNOVATIONS

### 2.1. THE PROBLEM AND PROPOSED SOLUTION

The Air Force and other DoD organizations are continuously developing new electronic components for

their weapons and other systems. In this effort they are in need of tools to aid in the automated design, or synthesis, of these components. One component of particular importance is the computer processor.

The rapid design of correct, efficient electronic hardware remains a difficult task despite advancements in design automation. This is even more the case for optimized, specialized hardware, because work in design automation has focused on general design problems rather than domain-specific ones, and has concentrated on reducing design time rather than increasing application specific hardware performance.

In general, the successful efforts in electronic computer aided design (ECAD) have been at the lower levels of design -- printed circuit board layout, gate array routing, standard cell placement, physical layout, and logic synthesis. The need remains for performance oriented higher level tools.

The higher levels of electronic computer aided design have been under study for a decade ([HLVS], [Survey], [SiliComp]), a research area that has come to be known as high level synthesis. High level synthesis offers the usual advantages of reduced design time through decreased human involvement in the design process (including decreased numbers of design errors through automation).

The general problem faced by high level synthesis is the usual one of producing designs that are as good (as small, as fast) as those produced by less automated techniques. Various optimizations developed for programming language compilation have been applied (and extended), with mixed success, to high level synthesis.

Recently a new optimization technique has been developed [Viper] which attempts to automate the hardware design process by automating the approach used to design RISC microprocessors. At its most basic, this technique consists of optimizing a design by optimizing for common cases, at the potential expense of infrequent ones, a common technique used by practicing engineers. This technique has been implemented in a research prototype tool, with suggestive results, but has not been widely tested and is not generally available in a usable tool.

The Phase I research reported here involved proof-of-concept tests of the synthesis technique, and the evaluation of the strengths and weaknesses of the synthesis system. During Phase II a robust tool will be constructed, evaluated, and demonstrated. Furthermore, reference manuals and user guides will be developed and training provided to USAF personnel in its use.

If successful, this tool would be valuable, specifically, for generating application-specific microprocessor implementations, and, more generally, application-specific hardware. Its significance would be in the rapidity with which it could generate error free, high performance hardware.

It would be of particular value to the Air Force, where component performance, design correctness, and design time of electronic components in weapons systems are all important. It would have more general value to anyone implementing a hardware system who desires to optimize all three of those factors.

## **2.2. INNOVATIONS**

The research performed here applies a new processor synthesis technique, based on the methods of RISC computer design, to the synthesis of the commercially available SPARC processor. This technique automatically generates, from a single instruction set architecture specification, different implementations optimized for different applications, with the different implementations based on the instruction frequencies of the applications. The system used in this Phase I effort contains the following innovations:

- The design automation tool operates at a higher level of abstraction, and thus is easier to use, provides error-free designs more rapidly, and handles larger designs than currently available CAD tools.
- Input to the tool is a high-level instruction set architecture, along with the instruction frequencies of hosted applications. Both of these inputs are natural and reasonable for the design domain; the instruction frequencies are completely application specific.
- The output of the tool is specialized processors tailored to specific applications, better optimized for those applications than general purpose ones.

### 3. PHASE I TECHNICAL OBJECTIVES

In general, current high level hardware synthesis techniques do not optimize hardware for specific applications. The general goal of this research is to explore a new technique for synthesizing specialized optimized electronic hardware.

Simply put, the new technique automates the Reduced Instruction Set Computer (RISC) design process. Prior to Phase I, the technique had been used to synthesize small microprocessors, and preliminary results were obtained, motivating further tests. In Phase I, larger scale tests with commercial microprocessors were performed. In Phase II a general purpose, robust tool will be constructed.

The specific Phase I research objectives were:

- Develop a SPARC processor specification in a format suitable for input to a prototype high level hardware synthesis system.
- Synthesize various microprocessor implementations of the SPARC, using the high level synthesis optimization techniques developed in [Viper] to explore the design space.
- Evaluate VHDL as a hardware input specification language, and as a mechanism for defining hardware components in the library of components available to the synthesis process.
- Prepare and submit a Phase I final report.

Note that these objectives have changed slightly from what was originally proposed. The Air Force's 1750A processor had been proposed as the test processor; at the Air Force's suggestion that was changed to the SPARC. Another proposed objective was the determination of an appropriate long-term hardware input specification language; due to the Air Force's interest in VHDL this became an extended examination of VHDL as both an input and library definition language (which subsumed another objective).

The results of Phase I -- a SPARC processor specification and resulting synthesized implementations, and an evaluation of what would be required for producing a robust and generalized synthesis tool -- should partially demonstrate the utility of high level synthesis techniques in the domain of Air Force applications, and should provide a road map to the release of a generally valuable tool.

If a Phase II is approved, the Phase II objectives are:

- Implement a robust and easily maintainable high level synthesis tool in C++.
- Employ VHDL as an input and library specification language, acquiring and customizing the necessary parser front end.
- Develop a component library (or libraries) of generally available components. (It is anticipated that users will also want to provide their own libraries.)
- Develop a set of Air Force oriented test cases and examples.

The results of Phase II -- a robust synthesis system, along with Air Force oriented test cases and demonstrations -- should prove the utility of the tool to the Air Force, and provide valuable examples to potential commercial users of the tool.

If the results of Phase II are substantial enough, they should form a solid basis for a Phase III commercialization effort, which would include long term tool support and marketing (including demonstrations at the Design Automation Conference).

### 4. TECHNICAL APPROACH

During Phase I the prototype synthesis tool has been extensively evaluated, and development of it into a generally usable tool has been planned. During Phase II the robust tool will be constructed, evaluated and demonstrated in application areas of most significance to the USAF. Furthermore, reference manuals and user guides will be developed and training provided to USAF personnel in its use. During Phase III commercialization, the tool will be integrated into widely used commercial and industrial design environments, readied for market and beta tested. Production and after-sales support plans will be developed.

## 4.1. TECHNICAL BACKGROUND

### 4.1.1. Introduction

Traditional, human-intensive methods of electronic hardware design are relatively slow and somewhat error prone. This has led to the explosive growth of the electronic computer aided design (ECAD) industry, and the proliferation of computer aided design (CAD) tools. Integrated circuits, in particular, have slow and expensive design iterations (due to mask generation and chip fabrication), and are hard to debug (it is much harder to probe them than a printed circuit board), and have thus been the focus of CAD tool development. Additionally, hardware designers have lacked the development tools possessed by software designers, who supported their own activities with tools such as compilers, debuggers, libraries, and software engineering environments.

The hardware designer's lot has been improving, however, with cell and chip libraries, routing tools, placement tools, better simulation environments (for the VHSIC Hardware Design Language (VHDL) and Verilog in particular), and design management support (often integrated into new and evolving CAD tool frameworks).

In general these tools have helped free designers from the necessity of manually managing the large quantities of detail needed to completely specify a hardware design. The focus of the tools, though, has been the lower levels of integrated circuit design. Higher level, system and application oriented optimization tools (analogous to optimizing compilers for programming languages) are lacking.

The work performed here takes advantage of recent research involving higher level optimization tools to develop a high level tool that will rapidly produce application specific hardware.

Consider an example. It is the instruction set architecture definition of a simple machine, defining the operation of microprocessor instructions. Individual instruction specific clauses are contained in a recursive instruction executing definition (during synthesis this tail recursion is converted to iteration).

```
run :-  
    fetch,  
    access(memDR, opcode, OP),  
    execute(OP),  
    run.  
run :- true.
```

The machine is composed of a fetch phase and an execute phase, which are recursively evaluated until one fails. The machine has four registers, a program counter (pc), an accumulator (ac), a memory address register (memAR), and a memory data register (memDR). The memDR has two fields, opcode and address.

The fetch phase is defined as a clause that retrieves an instruction from memory and increments the PC.

```
fetch :-  
    access(pc, PC), set(memAR, PC),  
    mem_read,  
    access(pc, OldPC), NewPC is OldPC+1, set(pc, NewPC).
```

The instructions are defined in the following execute clauses.

```
execute(halt) :- !,  
    fail.  
execute(add) :- !,  
    access(memDR, address, X), set(memAR, X),  
    mem_read,  
    access(memDR, T), access(ac, AC), A is T+AC, set(ac, A).  
execute(and) :- !,  
    access(memDR, address, X), set(memAR, X),  
    mem_read,  
    access(memDR, T), access(ac, AC), A is T & AC, set(ac, A).  
execute(shr) :- !,  
    access(ac, AC), A is AC>>1, set(ac, A).  
execute(load) :- !,  
    access(memDR, address, X), set(memAR, X),
```

```

    mem_read,
    access(memDR, T), set(ac, T).
execute(stor) :- !,
    access(memDR, address, X), set(memAR, X),
    access(ac, T), set(memDR, T),
    mem_write.
execute(jump) :- !,
    access(memDR, address, T), set(pc, T).
execute(brn) :-
    access(ac, AC), AC < 0, !,
    access(memDR, address, T), set(pc, T).
execute(brn) :- !,
    true.

```

This specification itself is relatively abstract, compared to many hardware specification and simulation languages. Explicit concurrency, timing, and connectivity (buses) are not present. It can be used as input to a process that automatically synthesizes hardware.

#### 4.1.2. High-Level Hardware Synthesis

High-level hardware synthesis is composed of three basic tasks (see [Survey] and [Tutorial]):

- translation of a behavioral specification, written in a hardware description or programming language, into an internal representation;
- scheduling of operations, which assigns each operator in the behavioral specification, such as "+", to a hardware time step, or cycle (synchronous hardware is assumed); and
- allocation of hardware elements, which assigns each operator to a piece of hardware, a "+" to an adder, for example (this includes both the selection of hardware elements and the mapping of operations to those elements).

These tasks are performed in the context of performance and resource constraints, with performance constraints usually expressed in terms of speed or delay, and resource constraints in terms of chip area.

#### 4.1.3. Translation

This task is essentially programming language compilation, from lexical input to intermediate representation, and is well understood (see [Dragon]). Most compiler optimizations used at this level, such as dead code elimination, for example, apply to high-level hardware synthesis. Any effective synthesis system must employ some of these common optimizations.

Less common compiler optimizations, particularly those used by vectorizing and parallelizing compilers, such as loop unrolling, can also be applied (see [Dragon]). These optimizations are characterized by code motion between basic blocks, and are, in general, large scale transformations. In high-level synthesis, such transformations are performed in connection with scheduling and related scheduling optimizations.

#### 4.1.4. Scheduling

The primary goal in scheduling is to balance higher performance (greater speed through greater concurrency) with lower cost (limited resources). In general, greater concurrency requires greater resources, which permit more operations to be performed in parallel. Hence scheduling is dependent on resource constraints and is thus affected by allocation.

Scheduling methods can be divided into two types. The first type always operates within the confines of basic blocks (in the compiler sense, a basic block being a sequence of consecutive statements in which flow of control only enters at the beginning and only leaves at the end). The second type moves operations between basic blocks, in an effort to increase concurrency. The second type often must duplicate operations in order to preserve correctness, further increasing cost for an added increase in performance.

### 4.1.5. Allocation

The primary goal of allocation is to generate cost effective data paths. The key to achieving this goal is sharing hardware -- having several behavioral operators use the same functional unit.

Virtually all allocation techniques attempt to produce minimal hardware within cost (area) and delay (critical path) constraints. The techniques differ in how they determine minimal cost.

## 4.2. ALTERNATE APPROACHES

The process of high level synthesis is a process of optimization, of generating the minimum amount of hardware needed with as much speed as possible, trading off hardware resources for hardware speed.

This optimization problem is a complex one, with several computationally difficult (NP-complete) subproblems. Thus there is no efficient algorithm that solves the problem, but rather a number of approximation techniques.

A number of optimization techniques have been developed for high level synthesis (Refer to [Viper] for a detailed discussion of these techniques). Many of them are based on mathematical methods that try to minimize hardware by maximizing hardware sharing.

High performance designs are possible, using application-driven optimization techniques developed for high level synthesis. These techniques are based on the observation, which has also motivated RISC-based computer design, that, in a given hardware system, some parts of the hardware are used more often than others, and that the hardware should be optimized for these common uses.

In particular, a specific application program will use some computer instructions more than others. A processor optimized for that application will have optimized implementations of those commonly used instructions.

RISC computers are generally based on the further observation that most applications use only a few, simple instructions. Hence, processors optimized for those applications will be simple.

The underlying design principle of optimizing for common cases, however, does not per se preclude complex instructions, if they are used frequently.

## 4.3. ADOPTED APPROACH

The general technical approach proposed here is that of high level hardware synthesis. In general it is promising because

- it operates at a high level of abstraction, saving design time and reducing the designer's exposure to detail, and
- it can optimize hardware implementations for specific applications.

A complete introduction to high level synthesis and its component tasks can be found in [Tutorial] and [Viper].

High level hardware synthesis consists of the generation of hardware structure from a high level specification of the hardware's behavior. Specifically:

**Input:** The input to high level synthesis is a "behavioral" specification, written in a hardware description language (similar to, and in some cases based on, a programming language). The input behavioral specification describes actions using programming language-like operators, such as "+".

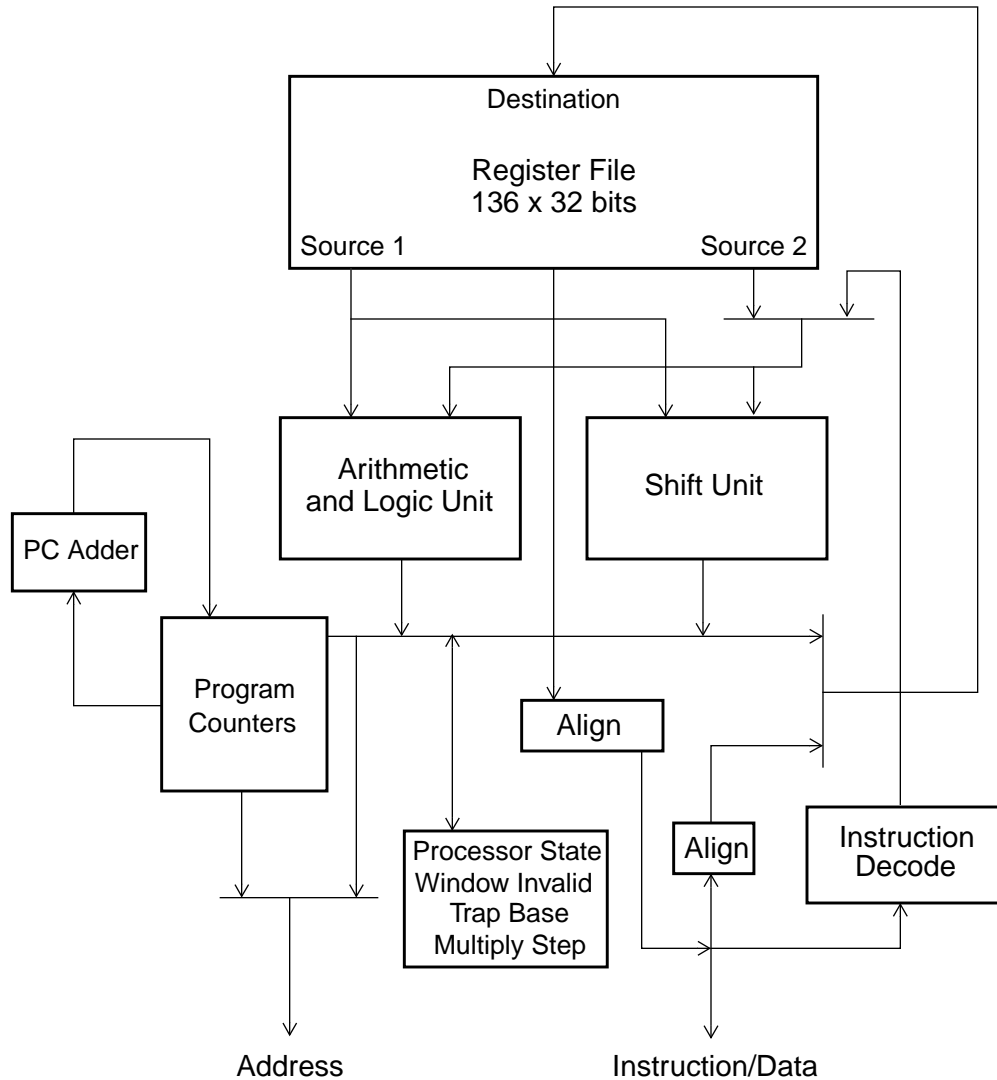
**Output:** The synthesis process creates netlists of hardware components, such as adders, which implement the desired actions, along with controlling logic, often in the form of a finite state machine.

For example, the behavior of a single computer instruction, an add instruction, is defined (see [SPARC-User], page 6-7) via:

$$[rd] \leftarrow r[rs1] + (r[rs2] \text{ or sign extnd(simm13)})$$

This adds the contents of two source registers and stores the result in a destination register. (A complete processor specification would include the definition of all the processor's instructions.) This behavioral description does not define how the addition is performed, or how the values in the registers are manipulated (how buses are used to transfer data between the registers). It is the task of high level synthesis to define such details.

The following figure shows what high level synthesis produces: a complete processor specification (including all instructions) -- a set of connected hardware components implementing the processor (the figure does not show all the controlling logic). This is the Cypress CY7C601 SPARC implementation's integer unit block diagram (from [SPARC-User], page 2-1). The diagram illustrates the implementation's commitment to a basic set of hardware elements and interconnections.



The optimization process employed here is in practice somewhat complicated [Viper], but it is based on the following three principles:

- Given constraints on how much hardware can be allocated, process instructions in the input specification in order of importance, allocating hardware for the most important operations first.
- Do not allocate hardware for unused instructions.
- Transform the specification using a well known method for microcode compilation called trace scheduling ([Trace]) that can increase the speed of execution of common instructions by increasing the potential parallelism in such instructions.

Automating this optimizing design strategy for high level synthesis requires more input to the synthesis process than just the input specification described above. It also requires weights indicating the relative importance of different parts of the specification. In the case of a microprocessor, this is simply the frequency with which each

instruction is executed; the more frequently executed instructions will have higher weights and will thus be more important. An artificial set of instruction frequencies, used in experiments, appears below.

```
% 40%
count(add, 25).
count(and, 15).
% 30%
count(load, 20).
count(stor, 10).
% 30%
count(brn, 18).
count(shr, 6).
count(jump, 5).
count(halt, 1).
```

A prototype high level synthesis system has been constructed that generates optimized processor implementations ([Viper-Intro], [Viper]). Prior to this Phase I it had been used to generate application specific versions of a general purpose processor ([BAM], [BAM-Manual]).

The system takes as input a microprocessor specification, the instruction frequencies of a compiled application program run with characteristic input data, and, using a library of hardware components from which to build the processor, produces as output a processor implementation. The system was developed as part of a larger CAD system ([ASP-Intro], [ASP-Layers], [ASP-Prototype]).

The system has thus far been used to generate microprocessors, but that orientation is not inherent to the system. It can operate on any high level hardware specification.

The essence of this Phase I has been to further test these optimization techniques on a more complicated, modern, commercially available microprocessor and determine the value of and effort needed to produce a robust, commercially viable high level synthesis tool.

## 5. TECHNICAL IMPLEMENTATION

This section presents information about the Viper high level synthesis system used in the Phase I effort. First its input language is described. Then its operation is briefly presented. A more complete discussion can be found in [Viper].

### 5.1. Specifying Hardware in Prolog

This subsection presents an introduction to hardware specification in Prolog. The advantages and drawbacks of using Prolog for hardware specification are in general the same as those for any programming language used for behavioral hardware specification. Its advantages are that it is familiar, it is rooted in a simple, general paradigm, and it is easily extended for hardware specification with built-ins. Its limitations all relate to the need to specify hardware details -- the representation of state (registers), and low-level features (bit fields, priority interrupts, efficient bit testing, efficient operators, and interfaces).

The microprocessor specifications that serve as input to the Viper hardware synthesis system are written in a subset of standard Prolog that roughly corresponds to the descriptive power of ISPS ([ISPS]). Specifications in this subset can be both simulated and synthesized.

The language has been both restricted and extended (with built-ins supporting such constructs as architected registers, bit fields, and a memory interface) to support hardware specification.

This level of hardware specification is designed to be the lowest level that can still be executed by a Prolog interpreter. It is essentially register transfer level computation performed in the context of Prolog control structures.

#### 5.1.1. Restrictions

Given the microprocessor specification domain of Viper, multiple asynchronous finite state machines, explicit parallelism, and detailed off-chip interface descriptions are not supported. The specification domain is also constrained by Viper's pragmatic purpose (and reason for existence) as a synthesis system. Specifications must be

effectively realizable in hardware.

The Prolog restrictions in particular are that:

- the specification must be deterministic (with only shallow backtracking),
- it can contain no lists or structures,
- it must be only tail-recursive, without true recursion, and
- it can use only a limited set of built-ins.

It is also assumed that: procedures do not fail; in a case, all arms are tagged with only one literal in the first position; in a case, all arms with the same tag are contiguous.

### 5.1.2. Extensions

On the other hand, the system also extends the standard Prolog built-in set, both to relax the above restrictions somewhat in a controlled way, and to support hardware-specific operations.

The extensions consist of three classes:

- support for maintaining global state (in the form of registers and register files),
- additional hardware-oriented operators (such as add with carry), and
- support for system functions (such as interfaces and memory).

### 5.1.3. Basic Extensions and Specifications

This subsection describes registers, fields, and memory, and their simulation.

Registers and fields are special in that they contain global state information. They also have definite bit widths and can overlap. Standard Prolog does not model objects with definite widths, nor does its single assignment nature support overlapping values.

Viper allows the user to declare registers and fields with the constructs

```
stateRegister(<name>, <width>).
stateField(<field-name>, <register-name>, <field-position>).
```

Thus every field is positioned within a specified register. More will be said about *<field-position>* below, which can have one of two forms.

Registers and fields are referenced with the *set* and *access* built-ins, which have the format

```
access(<register-name>, <Prolog-variable>), ...
set(<register-name>, <Prolog-variable>), ...
access(<register-name>, <field-name>, <Prolog-variable>), ...
set(<register-name>, <field-name>, <Prolog-variable>), ...
```

The access built-ins bind the values in the named registers and fields to the given Prolog variables, and the set built-ins change the register and field values.

A specification's state, when running, is contained in its set of registers. Simulation oriented built-ins are provided for creating and examining register sets.

A final additional set of built-ins provides a memory interface. This memory design frame requires that a memory address register and memory data register be declared; with those registers it performs read and write functions.

For example, an instruction fetch written in ISPS

```
memAR := pc;
mem_read;
pc := pc + 1;
```

would be written in Prolog as

```
access(pc, PC), set(memAR, PC),
mem_read,
access(pc, OldPC), NewPC is OldPC+1, set(pc, NewPC).
```

The specification can either be synthesized, or simulated with the aid of a package that provides the necessary built-ins.

Note that the total state -- the collection of registers -- is implicit in the specification, and is not an explicit structure that can be referenced by the user. If it were explicit the user could then refer to multiple states, which would be useful for temporal reasoning but difficult to implement.

#### 5.1.4. Extensions: Registers

Built-ins are provided for creating and manipulating registers and bit fields with specified widths, and for managing the state those registers contain. These built-ins were introduced above, and consist of:

- *stateRegister*,
- *stateField*,
- *access*,
- *set*, and
- *test*.

The last one, *test*, has the same form as *set* and *access*, and is a version of *access* that expects its value field to be bound, and succeeds if that bound value is equal to the accessed value.

#### 5.1.5. Extensions: Register Files

Viper provides built-ins for creating and referencing register files (indexed arrays of registers).

Register files are declared in a manner similar to individual registers, with an added size parameter, indicating the number of registers in the file. The declaration has the form

```
stateRegisterSet(<name>, <width>, <size>).
```

Thus the declaration

```
stateRegisterSet(r, 32, 16).
```

would create 16 32-bit registers.

Register file references involve two quantities, the value to be read or written, and the index of the register to be modified. Two built-ins are required. These built-ins take as arguments a port, an index, and a value, and read or write the value in one cycle. The built-ins have the form

```
rval(port(<file-name>, <port-name>), <index>, <value>), ...
% read value
wval(port(<file-name>, <port-name>), <index>, <value>), ...
% write value
```

For example, register 10 in register file *r* is read via

```
rval(port(r, read), 10, ReadValue), ...
```

and is written using

```
wval(port(r, write), 10, WriteValue), ...
```

#### 5.1.6. Extensions: Operators

Additional operators have been defined, to provide hardware oriented functionality. These operators can be constructed out of regular Prolog (the built-ins that simulate them obviously are), but they are defined explicitly so that Viper does not have to analyze the specification and determine, for example, that two adds, with one operand a one bit field, can be done as an add with carry.

In general, to produce high quality output, each specification requires a small collection of specially tailored operators.

### 5.1.7. Extensions: System Support

Additional built-ins have been defined to provide support for system-level functions, primarily in the form of interfaces. These interfaces are simple and stylized, and Viper has the capacity to generate the hardware associated with them.

The most fundamental interface is the one to the memory system. This interface, and its simulation, have been introduced above. Both the simulator and the synthesizer process *mem\_read* and *mem\_write* built-ins, and assume the proper data has been put into the registers named, by convention, *memAR* and *memDR*. The memory system also supports ported memory, invoked by the built-ins *mem\_read(<port-number>)* and *mem\_write(<port-number>)*. These reference similarly indexed address and data registers, *memAR(<port-number>)* and *memDR(<port-number>)*.

There is also a bitwise interface to lines that can be tested and set synchronously. These lines are declared with the *stateInterface* declaration, which defines the name of the bit line being declared.

## 5.2. The Viper Synthesis System

A general principle of design is to optimize for common cases. Within the context of global requirements and constraints, improving the performance of commonly occurring cases often improves the overall performance of a complete design. The successful application of this principle requires that common cases be identified, and that performance be effectively measured.

The Viper synthesizer applies this principle to the automated high-level synthesis of hardware, microprocessors in particular. For microprocessors, commonly executed instructions (common cases) are identified with instruction frequency statistics, and performance is measured in terms of instructions executed per unit of time. In general, Viper uses instruction frequencies to drive design choices. Such choices are reflected in the order in which objects are processed, and in weights associated with possible design elements.

Viper is a complete high-level synthesis system. It was primarily constructed to synthesize microprocessors rapidly -- to be used as a tool for architectural exploration. It was designed to operate without user interaction. It was also designed to reflect an architect's perspective on synthesis, particularly in its use of instruction frequency statistics. It uses Prolog for specification and implementation.

In general, high-level synthesis in Viper follows the form described above in Technical Background. Viper operates on input specifications written in the subset of executable Prolog described above. The output of Viper is a conventional collection of connected data path elements and a controlling finite state machine.

An extensive discussion of the implementation of Viper, along with source code, can be found in [Viper].

### 5.2.1. Scheduling In Viper

Viper uses a general, efficient, inter-block usage driven scheduler. In particular, high-level synthesis is similar to very long instruction word (VLIW) compilation (see [Bulldog]), with the instruction word width (the available resources) not fixed in advance. Viper applies this inter-block trace scheduling technique developed for VLIW compilation to high-level synthesis.

### 5.2.2. Allocation In Viper

Minimizing data path cost is not necessarily desirable when designing for performance (that is, overall execution speed). Commonly executed operations are more important than infrequent ones. Viper uses a greedy, iterative allocation technique that gives priority to common operations, allocating resources to them first.

### 5.2.3. The Operation of Viper

Viper has three fundamental modes of operation, basic, reducing, and optimizing.

- In the basic mode, Prolog specifications are translated into register transfer level representations, which are used for interrelated hardware scheduling and allocation, which in turn produce control and data paths.
- The reducing mode is similar to the basic mode, except that the hardware needed by seldom-used instructions (defined by a cutoff specified by the user) is omitted from the synthesized design.
- The optimizing mode uses trace scheduling to modify the intermediate RTL to improve the throughput of the design.

These modes are incorporated into the operation of Viper, which during synthesis performs the following steps:

- It translates the Prolog input specification into a data flow graph and a control flow graph.
- It converts the data flow graph into an equivalent set of register transfers.
- It computes instruction frequencies.
- It optionally trace schedules, using instruction frequencies.
- It computes preliminary as-soon-as-possible dependency-based intra-block schedules.
- It performs a preliminary allocation of functional units, guided by instruction frequencies.
- It performs a final allocation and scheduling, again guided by instruction frequencies, and by global resource constraints and frequency cutoffs (if any) supplied by the user.

#### 5.2.4. Measuring Results

The primary metrics conventionally applied to microprocessor implementations, and used with Viper to evaluate the quality of synthesized designs, are relative chip area and speed. Speed can be described in a number of ways (including clock rate and instructions executed per second); a well-known performance metric is employed here, cycles per instruction (CPI). The Viper system includes tools that compute CPI metrics and that measure the approximate size of control paths.

## 6. RESEARCH RESULTS

### 6.1. Translating ISP SPARC into Prolog SPARC

The SPARC architecture is defined behaviorally in ISP in Appendix C (pages 151-187) of Version 8 of *The SPARC Architecture Manual* [SPARC]. Developing a version of the SPARC specification in Prolog, suitable for synthesis, thus consisted of translating that ISP specification into Prolog. This was performed in three stages: converting the register definitions, translating the overall structure of the specification (trap-fetch-execute), and converting individual instructions.

Note that the floating point and multiprocessing components of the SPARC architecture were not translated.

See Technical Implementation above for a discussion of hardware specification in Prolog.

#### 6.1.1. Basic Translation

The translation of the SPARC specification into Prolog was performed in two stages. In the first stage, the essence of the ISP was captured in Prolog. In the second stage, the ability of the synthesis system to efficiently implement various constructs (especially control constructs and operators) was taken into consideration, and changes to the specification, the library, and the synthesis system itself were contemplated.

In the first stage the specification components were translated in sequence. The actual translations are presented in an appendix. The components, in translation order, are:

- architected register definitions;
- implementation register definitions;
- interface definitions;

- the trap-fetch-execute loop;
- trap handling;
- trap execution;
- instruction fetch;
- general instruction execution;
- updating the program counter;
- arithmetic, logical, and shift instructions;
- load and store instructions;
- call, jump, and link instructions;
- save and restore instructions,
- return from trap instructions; and
- branch and trap on condition code instructions.

### 6.1.2. Improving the Specification

After the specification was completed, it was reviewed for possible improvement and optimization. The actual improvements are presented in an appendix. The improvements include:

- trap handling dispatch;
- register windows;
- conditional sources and destinations in operands;
- sign extension; and
- conditions codes.

## 6.2. Synthesis of SPARC

Highlights of the Viper synthesis system are discussed above in Technical Implementation. A complete discussion can be found in [Viper].

### 6.2.1. Synthesis Experiments

To investigate the quality of Viper's various synthesis modes, a sequence of SPARC implementations was synthesized from the single Prolog SPARC specification presented above.

- A pure baseline implementation was synthesized, using the basic mode and uniform instruction weights.
- A gcc baseline version was synthesized, the same as the first version but using the gcc SPECmark instruction weights (see chapters 6 and 9 in [Viper] for a discussion of these weights) to compute the CPI metric.
- An eqntott baseline version was synthesized, similar to the first version but using the eqntott SPECmark instruction weights to compute the CPI metric.
- A high speed version was synthesized with a 30 nanosecond cycle time constraint, to generate a high speed data path implementation.
- A 0.1% reduced version was synthesized, in which the instruction use cutoff was 0.1% (the hardware needed only by instructions used less frequently was omitted). The gcc instruction weights were used. This was done to see how much smaller and faster such a reduced processor would be.

- A 1% reduced version was synthesized, the same as the 0.1% version, but with a 1% cutoff. This was done to compare with the 0.1% version.
- A second 1% reduced version was synthesized using the eqntott instruction weights. This was done to compare with the 1% gcc version.
- An optimized version was synthesized, using the gcc instruction weights and trace scheduling. This was done to evaluate trace scheduling.

All versions, unless otherwise noted, were synthesized with a 100 nanosecond cycle time constraint.

### 6.2.1.1. The Basic Uniform Version

The basic uniform version has: uniform instruction weights, all instructions synthesized, and no trace scheduling.

The major data path functional units allocated by Viper, as items in Prolog (including name, type, delay in nanoseconds, and area in square microns), were:

```
elem(o(alu,1), alu(ripple), 40.9, 280800).
elem(o(alu,2), alu(ripple), 40.9, 280800).
elem(o(cmp,1), cmp, 8.5, 56000).
elem(o(shf,1), shf, 17.7, 819840).
```

The data path operations (including unit name), were:

```
elemFn(o(alu,1), add).
elemFn(o(alu,1), and).
elemFn(o(alu,1), dec).
elemFn(o(alu,1), inc).
elemFn(o(alu,1), or).
elemFn(o(alu,1), sub).
elemFn(o(alu,1), xor).
elemFn(o(alu,2), add).
elemFn(o(cmp,1), gt).
elemFn(o(cmp,1), iszero).
elemFn(o(shf,1), asr).
elemFn(o(shf,1), shl).
elemFn(o(shf,1), shr).
```

Most of the data path area was taken up by registers and register files.

```
elem(reg(pc), reg, 3.1, 118400).
elem(reg(np), reg, 3.1, 118400).
elem(reg(usr), reg, 3.1, 118400).
elem(reg(tbr), reg, 3.1, 118400).
elem(reg(wim), reg, 3.1, 118400).
elem(reg(inst), reg, 3.1, 118400).
elem(reg(p), reg, 3.1, 118400).
elem(reg(q), reg, 3.1, 118400).
elem(reg(tempCWP), reg, 3.1, 118400).
elem(reg(tempAddr), reg, 3.1, 118400).
elem(reg(tempMask), reg, 3.1, 118400).
elem(reg(memAE), reg, 3.1, 3700).
elem(reg(memDR), reg, 3.1, 118400).
elem(reg(memAR), reg, 3.1, 118400).
elem(reg(memBM), reg, 3.1, 14800).
elem(reg(memAS), reg, 3.1, 29600).
elem(regfile(r), regfile, 9.4, 1460000).
```

The tabulated synthesis results are:

SPARC V8	Basic Version
Total Area	4560664
Cycles Per Instruction	5.61474
Number of blocks	700
Number of cycles	709
Number of transfers	1444

### 6.2.1.2. The Basic gcc Version

The basic gcc version has: gcc instruction weights, all instructions synthesized, and no trace scheduling.

The tabulated synthesis results are:

SPARC V8	Simple gcc Version
Total Area	4560664
Cycles Per Instruction	5.58755
Number of distinct cycles	709
Number of register transfers	1444

The only difference between this version and the one above is the expected one, namely a slightly different CPI result due to different instruction weights.

### 6.2.1.3. The Basic eqntott Version

The basic eqntott version has: eqntott instruction weights, all instructions synthesized, and no trace scheduling.

The tabulated synthesis results are:

SPARC V8	Simple eqntott Version
Total Area	4560664
Cycles Per Instruction	5.58755
Number of distinct cycles	709
Number of register transfers	1444

This, like the gcc version above, is the same as the basic uniform version except for a slightly different CPI result due to different instruction weights.

### 6.2.1.4. The High Speed Data Path Version

The high speed version has: gcc instruction weights, all instructions synthesized, no trace scheduling, and a 30 nanosecond cycle constraint.

The only difference between this version and the one above is that the synthesized data path has two faster, larger ALUs (carry bypass instead of ripple carry):

```
elem(o(alu,1), alu(bypass), 20.26, 392000).
elem(o(alu,2), alu(bypass), 20.26, 392000).
```

The tabulated synthesis results are:

SPARC V8	30 nanosecond Version
Total Area	4783064
Cycles Per Instruction	5.58755
Number of distinct cycles	709
Number of register transfers	1444

### 6.2.1.5. The 0.1% gcc Version

The 0.1% gcc version has: gcc instruction weights, instructions used 0.1% or less not synthesized, and no trace scheduling.

The synthesized data path was virtually the same, with the same functional units.

The data path operations required were:

- elemFn(o(alu,1), add).
- elemFn(o(alu,1), and).
- elemFn(o(alu,1), dec).
- elemFn(o(alu,1), or).
- elemFn(o(alu,1), sub).
- elemFn(o(alu,2), add).
- elemFn(o(cmp,1), gt).
- elemFn(o(cmp,1), iszero).
- elemFn(o(shf,1), shl).

The tabulated synthesis results are:

SPARC V8	0.1% gcc Version
Total Area	4546012
Cycles Per Instruction	5.56577
Number of distinct cycles	427
Number of register transfers	670

### 6.2.1.6. The 1% gcc Version

The 1% gcc version has: gcc instruction weights, instructions used 1% or less not synthesized, and no trace scheduling.

Again, the synthesized data path was virtually the same, with the same functional units.

The data path operations (including unit), were:

- elemFn(o(alu,1), add).
- elemFn(o(alu,1), and).
- elemFn(o(alu,1), dec).
- elemFn(o(alu,1), or).
- elemFn(o(alu,2), or).
- elemFn(o(cmp,1), gt).
- elemFn(o(shf,1), shl).

The tabulated synthesis results are:

SPARC V8	1% gcc Version
Total Area	4544680
Cycles Per Instruction	5.47583
Number of distinct cycles	386
Number of register transfers	553

### 6.2.1.7. The 1% eqntott Version

The 1% eqntott version has: eqntott instruction weights, instructions used 1% or less not synthesized, and no trace scheduling.

The list of required instructions:

keep(call).  
 keep(jmpl).  
 keep(save).  
 keep(restore).  
 keep(rett).  
 keep(lda).  
 keep(sta).  
 keep(ta).  
 keep(tne).

The synthesized data path was substantially smaller:

elem(o(alu,2), alu(ripple), 40.9, 280800).  
 elem(o(alu,1), alu(ripple), 40.9, 280800).  
 elem(o(cmp,1), cmp, 8.5, 56000).

Additionally, the tempMask, wim, and memBM registers were omitted.

The data path operations (including unit), were:

elemFn(o(alu,1), add).  
 elemFn(o(alu,1), and).  
 elemFn(o(alu,1), dec).  
 elemFn(o(alu,1), sub).  
 elemFn(o(alu,2), add).  
 elemFn(o(cmp,1), gt).  
 elemFn(o(cmp,1), iszero).

The tabulated synthesis results are:

SPARC V8	1% eqntott Version
Total Area	3470576
Cycles Per Instruction	5.37412
Number of distinct cycles	73
Number of register transfers	138

### 6.2.1.8. The Trace-Scheduled gcc Version

The trace-scheduled version has: gcc instruction weights, all instructions synthesized, and trace scheduling.

The synthesized data path was again the same, with the same functional units.

The data path operations (including unit), were:

```

elemFn(o(alu,1), add).
elemFn(o(alu,1), addc).
elemFn(o(alu,1), and).
elemFn(o(alu,1), dec).
elemFn(o(alu,1), inc).
elemFn(o(alu,1), invert).
elemFn(o(alu,1), or).
elemFn(o(alu,1), sub).
elemFn(o(alu,1), subc).
elemFn(o(alu,1), xor).
elemFn(o(alu,2), add).
elemFn(o(cmp,1), equal).
elemFn(o(cmp,1), gt).
elemFn(o(shf,1), asr).
elemFn(o(shf,1), shl).
elemFn(o(shf,1), shr).

```

The tabulated synthesis results are:

SPARC V8	Trace Version
Total Area	4560664
Cycles Per Instruction	3.98386
Number of distinct cycles	1667
Number of register transfers	3543

The additional time taken by the trace scheduling optimizer was 3 1/2 hours.

### 6.3. Translating Prolog into VHDL

VHDL is a complex language with multiple capabilities. It is: a programming language; a simulation language; a language for describing hardware structure; and a language for specifying hardware timing characteristics. Recasting Viper to operate on VHDL meant choosing an appropriate subset of VHDL, one that roughly corresponded to the subset of Prolog used by Viper. The VHDL Cookbook, by Peter J. Ashenden [VHDL-Intro], and the VHDL Language Reference Manual [VHDL-STD] were used as references.

The starting point for these investigations was the Prolog SPARC specification, not the ISP of the original Sun specification, because the synthesizer operates on Prolog, and the basic operation of the synthesizer was to be preserved.

The translation proceeded in stages.

- The register definitions were translated, ultimately using VHDL alias statements.
- Interfaces were translated using VHDL signal declarations.
- Register files were converted to VHDL arrays.
- A canonical arithmetic instruction, add, was translated, ultimately using connection variables.
- The load and store instructions were translated using mechanisms developed for the add instruction.
- The main run loop was translated.
- Instruction dispatch and traps were translated using VHDL control structures.

Translation was relatively straightforward once the appropriate VHDL constructs were developed.

Details of the translation can be found in an appendix.

## 7. CONCLUSIONS AND RECOMMENDATIONS

This section summarizes the Phase I results, draws conclusions from those results, and makes some recommendations about future work.

### 7.1. SPARC Synthesis Result Summary

The following table summarizes the results of the different synthesis experiments performed with the SPARC specification and Viper:

Version	Basic	gcc	eqntott	30 ns	0.1% gcc	1% gcc	1% eqntott	Trace
Instruction Weight	uniform	gcc	eqntott	gcc	gcc	gcc	eqntott	gcc
Reduction	0%	0%	0%	0%	0.1%	1%	1%	0%
Other Synthesis Option	-	-	-	fast clock	-	-	-	trace sched
Total Area	4560664	4560664	4560664	4783064	4546012	4544680	3470576	4560664
Cycles Per Instruction	5.61474	5.58755	5.57457	5.58755	5.56577	5.47583	5.37412	3.98386
Number of distinct cycles	709	709	709	709	427	386	73	1667
Number of register transfers	1444	1444	1444	1444	670	553	138	3543

Highlighting these numbers, the following table shows the percentage differences between the basic uniform version and the other versions.

Version	gcc	eqntott	30 ns	0.1% gcc	1% gcc	Trace
Area benefit (cost)	0%	0%	(4.88%)	0.32%	0.35%	0%
CPI benefit	0.49%	0.72%	0.49%	0.88%	2.54%	40.94%
Transfer benefit (cost)	0%	0%	0%	54%	62%	(145%)

These results indicate:

- There is little difference in data paths between versions. The data paths are in fact relatively simple, and there is little room for difference.
- The reduced versions feature smaller control, but no other major differences -- except for the questionable eqntott results.
- Most of the smaller control benefit of the reduced versions comes at 0.1%.
- Trace scheduling improves throughput substantially with the same basic data path.
- The trace scheduling throughput improvement comes at the price of a proportionally larger increase in control size.

The important meta observation here is that once the SPARC specification was constructed, different processor implementations could be generated with a few minutes of setup and a few hours (see below) of synthesis time.

### 7.2. VHDL and Viper

A simple definition of the subset of VHDL synthesizable by Viper (driven by the need for specifications to be effectively realizable in hardware) is:

- a pre-defined set of data types, including field, storage, and connection data types, must be used;
- other types can appear only in library models;
- a pre-defined memory interface must be used;
- a limited set of built-in functions are available; and
- recursion is not allowed.

Performing the translations enumerated above in Research Results has led to some observations on using Viper for VHDL synthesis.

- The notion of a cycle -- access-operation-set -- permeates Viper specifications and Viper synthesis. Synthesizable VHDL can be written without access and set, but, to use Viper, ultimately that VHDL will be translated into some form using access and set.
- Prolog specifications have a relatively flat control structure. There are few levels of procedural nesting or conditionals. The Prolog procedure is the basic unit of conditional execution, and such procedures are not simple, local constructs. VHDL, in contrast, has a rich collection of control structures. These control structures make specifications simpler and more readable, but complicate translation by Viper.
- With structural VHDL, operations such as add with carry can be defined easily (in contrast with Prolog). Such functions will not be synthesized by Viper, however. Such library VHDL can serve as a simulation model, with, for example, an actual add function implemented in an optimized ALU.
- VHDL control structures also allow the easy expression of functions that can more efficiently be implemented with logic gates, as opposed to finite state machine states. Consider, for example, the selection of the second operand to the add instruction, which can either be a register or an immediate. This if-then is best implemented in gates, as opposed to a multiple state test. In VHDL such functions can be easily identified by the user and realized using logic synthesis.
- For register field data dependency analysis, the VHDL alias statements will have to be used to determine the actual registers where the data resides.

### 7.3. An Assessment of the Current Viper System

The current implementation of Viper was, in general, adequate for these experiments. Its weaknesses can be divided into three categories, bugs, library extensions, and performance.

#### 7.3.1. Viper Bugs

Coming into these experiments, the Viper system was surprisingly robust, with one major exception.

The exception was the component that prepares register transfers for trace scheduling (the first item in the trace scheduling action table below). It was quite limited, and, as originally written, could not handle the complexity of the SPARC design. It required substantial modification (it had to be extended to handle seven additional cases).

Otherwise, only one other bug was detected, that in the data path allocation code (again, exercised because of the complexity of the SPARC specification).

#### 7.3.2. Viper Library Elements

Each processor synthesized by Viper so far (the 6502, the BAM, and the SPARC) has required additional operators. This has been the case for two reasons. First, the set of operators in Prolog is not particularly rich, especially with respect to common hardware operations (there is no exclusive or, for example). Second, in order to produce implementations with good performance, some complex computations are best made into hardware modules, put in the library, and treated as single operators by Viper.

With the SPARC, add with carry and sign extension were added to the library. Because of the impact operators have on parsing and RTL optimization, modification of the Viper code that performs those functions was needed (five modules in all -- lib, opmap, scan, mort, and como -- were changed; see Appendix F in [Viper] for a discussion of adding operators to the system).

In a system retargeted to VHDL the most unpleasant aspects of this extension process would be obviated. In particular, VHDL's support of modularity and overloading would greatly simplify specification processing. Additionally, structural VHDL is a natural mechanism for representing connectivity.

### 7.3.3. Viper Performance

Because of the size of the SPARC specification, the speed of Viper became a factor, which it had not been in earlier experiments (see [Viper]).

A basic SPARC design takes about six hours to synthesize, and a trace scheduled design takes nine hours. The component actions and their times, on a Sun SPARCstation 1:

Basic Action	Elapsed Time	Prolog Used
translate Prolog to RTL	4:09:52	C-Prolog
optimize RTL	4:11	C-Prolog
compute probabilities	1:00:11	SICStus
schedule RTL	6:42	SICStus
compute data path needs	5:05	SICStus
create data path	3:15	SICStus
compute throughput	3:19	C-Prolog
estimate control size	4:04	C-Prolog

Trace scheduling actions (starting with the optimized RTL of a basic design) and their times:

Trace Scheduling Action	Elapsed Time	Prolog Used
convert RTL for trace scheduling	2:26	C-Prolog
trace schedule	29:18	SICStus
convert RTL for allocation	28:11	SICStus
optimize RTL	22:17	C-Prolog
compute probabilities	55:42	SICStus
schedule RTL	43:39	SICStus
compute data path needs	18:36	SICStus
create data path	6:45	SICStus
compute throughput	6:16	C-Prolog
estimate control size	17:43	C-Prolog

Given the known performance weaknesses of Prolog, and the fact that Viper is essentially a compiler and should exhibit performance similar to compilers for programming languages, it is reasonable to expect that a reimplementation of Viper in a more efficient language would speed up its operation considerably.

### 7.4. Overall Conclusions

The major findings of this project are:

- The Prolog-based hardware specification method defined in [Viper] was adequate to specify a modern RISC processor, meeting its design goal of being equivalent to ISP.
- Using VHDL for specification, once initial definitional issues were overcome, was substantially easier and more natural than using Prolog. VHDL's rich control structures made specifications much more readable, and the ability to specify hardware structure made library elements much easier to define.
- Defining a library of operators and procedures was necessary with both languages, in order to realize designs of reasonable quality. Such definitions are in fact the method used to tailor the Viper

synthesis process to a particular microprocessor design, by hand optimizing selected constructs.

- Non-pipelined designs were generated quickly and relatively easily.
- Enhanced designs (reduced and trace scheduled) were generated easily.
- Reduced designs featured smaller control paths, but not smaller data paths.
- The trace scheduled design was substantially faster, with a much larger control path.
- The Viper system was robust enough to handle the SPARC design, but it pushed the limits of the implementation. Performance in particular became a factor, which it had not been in earlier experiments (see [Viper]).

In sum, the current system is adequate for rapid prototyping non pipelined versions of commercial RISC microprocessors. It supports both automatic optimization (through trace scheduling and instruction set reduction) and manual improvement (via additions to the library).

The Prolog implementation of the Viper system has been adequate for these tests, but any further development will require a reimplementaion.

## 7.5. A Processor Design Aid

Viper was primarily constructed to synthesize high-quality microprocessors rapidly -- to be used as a tool for architectural exploration. It was designed to operate without user interaction. It was also designed to reflect an architect's perspective on synthesis, particularly in its use of instruction frequency statistics.

It has demonstrated its ability to generate reasonable implementations quickly given a specification and a tuned library.

This basic functionality is valuable and can be pursued further.

Viper has two weaknesses that spring from its original design goals. It was designed to operate without user interaction, and was designed to produce designs quickly, which meant de-emphasizing pipelining and its complexities. These weaknesses became obvious as Viper was used as a tool in this Phase I.

A new implementation can address these issues.

- The allocator can be easily modified to allow the human designer to partly specify and modify the data path (in a user-friendly manner). This would allow the designer more direct control over the data path, as opposed to indirectly affecting it by changing the library. (The capability to modify the library is additionally useful and would be retained.)
- The basic scheduler and allocator can now already work with partially specified schedules; this feature has been used to generate pipelined designs (see Chapter 13 in [Viper]). Such partial schedules are now difficult to specify -- the user-system interface should be improved.
- Feedback from Viper about the synthesized design can be improved. It should, for example, inform the user as to which instructions are not being implemented in reduced mode.
- In part because of the textual orientation of Prolog (lacking, for example, the graphic libraries of Visual C++), and in part because of the rapid prototyping nature of Viper development, Viper's user interface is primitive. The system should be able to display a graphical representation of the developing design, with which the user can interact. Structural components in particular, such as library elements and the data path, should be displayed graphically.
- Current support for pipelining could be improved. The rup tool, part of Viper (see chapter 13 in [Viper]), identifies common resource usage patterns in instruction definitions, patterns that suggest pipelining strategies. The tool was applied to the SPARC, and identified 519 potential resource usage patterns in various parts of the specification (as opposed to 17 for the BAM microprocessor). This tool could be enhanced by using classes of resources to reduce the number of potential patterns, and by detecting dependencies between patterns. The tool could also be

more useful if it could be applied interactively, with the user indicating potential shared resources and selecting promising patterns.

The above enhancements, in addition to a reimplementaion in C++ and a retargeting to VHDL, would lead to a valuable prototyping tool.

## 8. REFERENCES AND BIBLIOGRAPHY

[ASP-Intro]

"An Advanced Silicon Compiler in Prolog"; William R. Bush, Gino Cheng, Patrick C. McGeer, Alvin M. Despain; *1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1987, pp. 27-31.

[ASP-Layers]

"Layering Expertise in a Full-Range Hardware Synthesis System"; William R. Bush, Gino Cheng, Alvin M. Despain; *IFIP WG10.2 Working Conference on CAD Systems using AI Techniques*, June 1989.

[ASP-Prototype]

"A Prototype Silicon Compiler in Prolog"; William R. Bush, Gino Cheng, Patrick C. McGeer, Alvin M. Despain; *UC Berkeley CS Technical Report UCB/CSD 88/476*, December 1988.

[Bulldog]

*Bulldog: A Compiler for VLIW Architectures*; John R. Ellis; MIT Press, 1985.

[BAM]

"Fast Prolog with an Extended General Purpose Architecture"; Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, William R. Bush, Alvin M. Despain, Joan M. Pendleton, Tep Dobry; *Seventeenth International Symposium on Computer Architecture*, May 1990.

[BAM-Manual]

*The Berkeley Abstract Machine Instruction Manual*; Bill Bush, Mike Carlton, Alvin Despain, Ralph Haygood, Bruce Holmer, Barton Sano, Peter Van Roy, Charlie Burns, Joan Pendleton; 12 December 1989.

[Dragon]

*Compilers: Principles, Techniques, and Tools*; Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman; Addison-Wesley, 1988.

[HLVS]

*High-Level VLSI Synthesis*; Raul Camposano, Wayne Wolfe, editors; Kluwer Academic Publishers, 1991.

[ISPS]

"Instruction Set Processor Specifications (ISPS): The Notation and Its Applications"; Mario R. Barbacci; *IEEE Transactions on Computers*; January 1981; pp. 24-40.

[SiliComp]

*Silicon Compilation*; Daniel D. Gajski, editor; Addison-Wesley, 1988.

[SPARC]

*The SPARC Architecture Manual*; SPARC International, 1991.

[SPARC-User]

*SPARC RISC User's Guide*; Cypress Semiconductor; 1990.

[Survey]

*A Survey of High-Level Synthesis Systems*; Robert A. Walker, Raul Camposano; Kluwer Academic Publishers, 1991.

[Trace]

"Trace Scheduling: A Technique for Global Microcode Compaction"; Joseph A. Fisher; *IEEE Transactions on Computers*; July 1981; pp. 478-490.

[Tutorial]

"Tutorial on High-Level Synthesis"; Michael C. McFarland, Alice C. Parker, Raul Camposano; *25th Design*

*Automation Conference*, 1988; pp. 330-336.

[Viper]

*The High-Level Synthesis of Microprocessors Using Instruction Frequency Statistics*; William R. Bush; Ph.D. dissertation, University of California, Berkeley; and *UC Berkeley Electronics Research Laboratory Memorandum No. UCB/ERL M92/109*, May 1992.

[Viper-Intro]

"Experience with Prolog as a Hardware Specification Language"; William R. Bush, Gino Cheng, Patrick C. McGeer, Alvin M. Despain; *Fourth Symposium on Logic Programming*, September 1987, pp. 490-498.

[VHDL-STD]

*IEEE Standard VHDL Language Reference Manual*; IEEE Std 1076-1987, 1988.

[VHDL-Intro]

*The VHDL Cookbook, First Edition*; Peter J. Ashenden; University of Adelaide, South Australia, 1990.

## 9. Appendix: The Prolog SPARC Specificaiton

### 9.1. Architected Register Definitions

The SPARC processor features six registers and three register files (defined in Appendix C.3). The registers are the Processor State register (psr), the Trap Base Register (tbr), the Window Invalid Mask Register (wim), the Y Register (y), the Program Counter (pc), and the Next Program Counter (npc). The register files are the Windowed Registers (r[i]), the Global Registers (g[i]), and the Ancillary State Registers (asr[i]). The psr and the tbr have a number of component fields.

The definitions of these registers were readily converted to Prolog using the Prolog constructs introduced above (Specifying Hardware In Prolog). The psr is the most complex, with numerous bit fields.

```
% Processor State Register
stateRegister(psr, 32).
stateField(psr, impl, (31-28)). % implementation id
stateField(psr, ver, (27-24)). % implementation version
stateField(psr, icc, (23-20)). % IU condition codes
stateField(psr, n, 23). % negative
stateField(psr, z, 22). % zero
stateField(psr, v, 21). % overflow
stateField(psr, c, 20). % carry
stateField(psr, reserved_PSR, (19-14)). % unused
stateField(psr, ec, 13). % enable CP
stateField(psr, ef, 12). % enable FP
stateField(psr, pil, (11-8)). % acceptable interrupt level
stateField(psr, s, 7). % supervisor mode
stateField(psr, ps, 6). % previous s value
stateField(psr, et, 5). % enable traps
stateField(psr, cwp, (4-0)). % current window pointer
```

The pc and npc registers are straightforward.

```
% Program Counter
stateRegister(pc, 32).

% Next Program Counter
stateRegister(npc, 32).
```

The windowed register file of the SPARC is somewhat complicated. It involves a movable pointer into an otherwise normal register file (the pointer is contained in the cwp field of the psr, above), which is used by the hardware as an index into the file on every register reference. Each window contains 16 registers (register specifiers are 4 bits), half of which are overlapped with its preceding window and half with its following window. The SPARC architecture requires that every implementation have at least 2 windows, and a maximum of 32. For the Phase I, the number of windows was set at 4, to keep the register file small in relation to the rest of the data path (to better illuminate

differences in data path area). Additionally, the window indexing mechanism was not specified behaviorally in Prolog, but implemented in the register file addressing logic.

The windowed register file was easily defined, but its implementation is subtle; these subtleties are captured in comments based on the ISP. The wim register, which specifies which windows can be used, was also defined.

```
% Windowed Registers
nwindows(4).
stateRegisterSet(r, 32, 64).
% *** note:
% *** two read and write ports (r1, r2)
% *** same-cycle indexing used
% *** Since R0 contains 0,
% wval(port(r, w), 0, N)
% set(port(r, w), 0), set(reg(r, w), N)
% *** suppresses storing to R0
% Windowed register reference r[n]:
% if (n = 0) then 0
% else if (1 <= n <= 7) then G[n]
% else R[((n-8)+(CWP*16)) modulo (16*NWINDOWS)]
% This is implemented by running psr.cwp into the register file
% address decode (implicitly part of the port builtin).

% Window Invalid Mask Register
stateRegister(wim, 32).
```

Other assorted registers are needed in the specification.

```
% Trap Base Register
stateRegister(tbr, 32).
stateField(tbr, tba, (31-12)).
stateField(tbr, tt, (11-4)).
stateField(tbr, zero, (3-0)).

% Y
stateRegister(y, 32).

% Global Registers
stateRegisterSet(g, 32, 7).

% Ancillary State Registers
stateRegisterSet(asr, 32, 31).
```

## 9.2. Implementation Register Definitions

In addition to these architected registers, some implementation registers were defined. Implementation dependent registers (16-31) of the asr register file could have been used, but, for clarity, separate individual registers were defined.

One important register (more precisely, set of flip-flops), p, holds the state of the processor. This register is implied in the ISP but not explicitly represented as such; its component bits appear as ISP variables.

```
% Implementation Registers

% the global state register and its bits
stateRegister(p, 32).
stateField(p, annul, 32).
stateField(p, cp_disabled, 31).
stateField(p, cp_exception, 30).
stateField(p, data_access_error, 29).
stateField(p, data_access_exception, 28).
stateField(p, data_store_error, 27).
stateField(p, division_by_zero, 26).
stateField(p, error_mode, 25).
```

```

stateField(p, execute_mode, 24).
stateField(p, fp_disabled, 23).
stateField(p, fp_exception, 22).
stateField(p, illegal_instruction, 21).
stateField(p, instruction_access_error, 20).
stateField(p, instruction_address_exception, 19).
stateField(p, mem_address_not_aligned, 18).
stateField(p, privileged_instruction, 17).
stateField(p, r_register_access_error, 16).
stateField(p, reset_mode, 15).
stateField(p, reset_trap, 14).
stateField(p, tag_overflow, 13).
stateField(p, trap, 12).
stateField(p, trap_instruction, 11).
stateField(p, unimplemented_FLUSH, 10).
stateField(p, window_overflow, 9).
stateField(p, window_underflow, 8).

```

The q register is used to hold trap and interrupt information.

```

% the interrupt and trap level register
stateRegister(q, 32).
stateField(q, interrupt_level, (13-10)).% <3:0> from bp_IRL
stateField(q, ticc_trap_type, (6-0)).% <6:0> from icc trap

```

Three temporary registers are used in computation.

```

% temporaries for SAVE and RESTORE, JMPL, RETT
stateRegister(tempAddr, 32).
stateField(tempAddr, byteAddr, (1-0)).
stateRegister(tempCWP, 32).
stateRegister(tempMask, 32).

```

An instruction register, with its decode fields (which are part of the architectural specification), is used to hold the results of instruction fetch for decoding. This register could have been integrated with the memory system, but for simplicity was defined separately.

```

% The instruction register and its fields
stateRegister(inst, 32).
stateField(inst, op, (31-30)).
stateField(inst, op2, (24-22)).
stateField(inst, op3, (24-19)).
stateField(inst, opf, (13-5)).
stateField(inst, opc, (13-5)).
stateField(inst, asi, (12-5)).
stateField(inst, i, 13).
stateField(inst, rd, (29-25)).
stateField(inst, a, 29).
stateField(inst, cond, (28-25)).
stateField(inst, rs1, (18-14)).
stateField(inst, rs2, (4-0)).
stateField(inst, simm13, (12-0)).
stateField(inst, shcnt, (4-0)).
stateField(inst, trapnum, (6-0)).
stateField(inst, disp30, (29-0)).
stateField(inst, disp22, (21-0)).

```

Note that the instruction fields are defined in Appendix C.4.

The memory interface registers were translated and defined (as per Appendix C.2).

```

% Memory system use in ISP:
%
```

```

% (load_data, MAE) <- memory_read(addr_space, address)
%
% MAE <- memory_write(addr_space, address, byte_mask, store_data)
%
% Memory system use in Prolog:
%
% set(memAS, Space),
% set(memAR, Addr),
% mem_read,
% access(memDR, Data), ... (memAE also available)
%
% set(memAS, Space),
% set(memAR, Addr),
% set(memBM, Mask),
% set(memDR, Data),
% mem_write, ... (memAE available)
%
stateRegister(memAR, 32).
stateField(memAR, doubleAddr, (2-0)).
stateField(memAR, wordAddr, (1-0)).
stateField(memAR, halfAddr, 0).
stateRegister(memDR, 32).
stateField(memDR, firstByte, (31-24)).
stateField(memDR, secondByte, (23-16)).
stateField(memDR, thirdByte, (15-8)).
stateField(memDR, fourthByte, (7-0)).
stateField(memDR, firstHalf, (31-16)).
stateField(memDR, secondHalf, (15-0)).
stateRegister(memAS, 8).
stateRegister(memBM, 4).
stateRegister(memAE, 1).

```

The byte, half word, word, and double word fields are used by the load and store instructions.

### 9.3. Interface Definitions

Off-chip interface lines were defined (from Appendix C.2).

```

stateInterface(bp_IRL, (3-0)).
stateInterface(bp_reset_in, 0).
stateInterface(pb_error, 0).
stateInterface(pb_block_ldst_word, 0).
stateInterface(pb_block_ldst_byte, 0).
stateInterface(bp_FPU_present, 0).
stateInterface(bp_FPU_exception, 0).
stateInterface(bp_FPU_cc, (1-0)).
stateInterface(bp_CP_present, 0).
stateInterface(bp_CP_exception, 0).
stateInterface(bp_CP_cc, (1-0)).

```

### 9.4. The Trap-Fetch-Execute Loop

The basic structure of the SPARC specification consists of (Appendices C.5 and C.6): 1) a reset test and wait loop, 2) an error test and wait loop, 3) trap execution, 4) instruction fetch, 5) instruction execution, and 6) program counter update. This was defined in Prolog.

The basic tail-recursive run loop, the heart of the specification, was defined.

```

run :-
    reset,
    error,

```

```

trap,
fetch,
execute,
run.
run :- true.

```

The reset loop, which loops while reset is high, was defined. It initializes the basic state of the processor.

```

reset :-
    test(bp_reset_in, 1), !,
    % initialize state
    set(p, reset_mode, 0),
    set(p, execute_mode, 1),
    set(p, trap, 1),
    set(p, reset_trap, 1),
    reset.
reset :- true.

```

The error loop, which loops while error is high, was defined. Error reset was also defined.

```

error :-
    test(p, error_mode, 1), !,
    errorReset,
    error.
error :- true.

errorReset :-
    test(bp_reset_in, 1), !,
    % initialize state
    set(p, error_mode, 0),
    set(p, reset_mode, 1),
    set(pb_error, 0).
errorReset :- !.

```

## 9.5. Trap Handling

The trap handler was defined. It sets the interrupt level, depending on the mode of the processor. It also causes the trap to be executed.

```

trap :-
    trapint,
    traptest.

% set the interrupt level
trapint :-
    test(psr, et, 1),
    access(bp_IRL, IRL),
    RL = 15, !,
    set(p, trap, 1),
    set(q, interrupt_level, IRL).
traptest :-
    access(psr, pil, PIL),
    access(bp_IRL, IRL),
    IRL > PIL, !,
    set(p, trap, 1),
    set(q, interrupt_level, IRL).
trapint :- !.

% if there is a trap pending, execute it
traptest :-
    test(p, trap, 1), !,

```

```
trapex.  
traptest :- !.
```

## 9.6. Trap Execution

Trap execution was defined. The handling of individual traps consists of (see Appendix C.8)

- identifying the trap,
- zeroing various state bits,
- and setting the program counter appropriately.

First, the root procedure including setting state bits:

```
trapex :-  
    % identify the trap  
    trapsel,  
    % zero state bits  
    set(p, trap, 0),  
    set(p, instruction_access_exception, 0),  
    set(p, illegal_instruction, 0),  
    set(p, privileged_instruction, 0),  
    set(p, fp_disabled, 0),  
    set(p, cp_disabled, 0),  
    set(p, window_overflow, 0),  
    set(p, window_underflow, 0),  
    set(p, mem_address_not_aligned, 0),  
    set(p, fp_exception, 0),  
    set(p, cp_exception, 0),  
    set(p, data_access_exception, 0),  
    set(p, tag_overflow, 0),  
    set(p, division_by_zero, 0),  
    set(p, trap_instruction, 0),  
    set(q, interrupt_level, 0),  
    % update the pc  
    trappc, !.
```

Second, trap identification:

```
% identify the trap  
trapsel :-  
    test(p, reset_trap, 1), !.  
trapsel :-  
    test(psr, et, 0), !,  
    set(p, execute_mode, 0),  
    set(p, error_mode, 0).  
trapsel :-  
    test(p, data_store_error, 1), !,  
    set(tbr, tt, tmask1).  
trapsel :-  
    test(p, instruction_access_error, 1), !,  
    set(tbr, tt, tmask2).  
... (several traps omitted for brevity)  
trapsel :-  
    % interrupt level > 0  
    access(q, interrupt_level, IL),  
    set(tbr, tt, IL).  
    % tt <- 0001 [] interrupt_level
```

Third, updating the pc -- this involves

- testing for errors and leaving the pc untouched in that case;
- setting the window pointer;

- saving the old pc (based on annulling); and
- setting the new pc (based on whether or not the trap was a reset):

```

% update the pc
trappc :-
    test(p, error_mode, 0), !,
    set(psr, et, 0),
    access(psr, s, OldS),
    set(psr, ps, OldS),
    trapCWP,
    trappcLast,
    set(psr, s, 1),
    trappcNext.
trappc :- !.

% set the current window pointer
trapCWP :-
    test(psr, cwp, 0), !,
    % NewCWP is NWindows - 1
    % nwindows = 4
    set(psr, cwp, 3).
trapCWP :-
    access(psr, cwp, OldCWP),
    NewCWP is OldCWP - 1,
    set(psr, cwp, NewCWP).

% save the old pc
trappcLast :-
    test(p, annul, 0), !,
    access(pc, OldPC),
    wval(port(r, w), 17, OldPC),
    access(npc, OldNPC),
    wval(port(r, w), 18, OldNPC).
trappcLast :-
    access(npc, OldPC),
    wval(port(r, w), 17, OldPC),
    OldNPC is OldPC + 4,
    wval(port(r, w), 18, OldNPC),
    set(p, annul, 0).

% set up the new pc
trappcNext :-
    test(p, reset_trap, 0), !,
    access(tbr, ThisPC),
    set(pc, ThisPC),
    NextPC is ThisPC + 4,
    set(npc, NextPC).
trappcNext :-
    set(pc, 0),
    set(npc, 4),
    set(p, reset_trap, 0).

```

## 9.7. Instruction Fetch

Instruction fetch was defined. It distinguishes between user mode and privileged instructions.

```

fetch :-
    test(psr, s, 0), !,
    set(memAS, 8),
    access(pc, Addr),

```

```

        set(memAR, Addr),
        mem_read.
fetch :-
    set(memAS, 9),
    access(pc, Addr),
    set(memAR, Addr),
    mem_read.

```

## 9.8. General Instruction Execution

General instruction execution was defined. It distinguishes between an annulled instruction (the SPARC architecture features one delay slot after jumps, with associated annulling), a bad instruction fetch address, and a normal instruction.

```

execute :-
    % annul
    test(p, annul, 1), !,
    set(p, annul, 0),
    access(npc, PC),
    set(pc, PC),
    NPC is PC + 4,
    set(npc, NPC).
execute :-
    % instruction read exception
    test(memAE, 1), !,
    set(p, trap, 1),
    set(p, instruction_address_exception, 1).
execute :-
    access(inst, op, OPex),
    disptest(OPex),
    dispatch(OPex),
    access(inst, op, OPup),
    updatePC(OPup).

```

The code was defined that performs preliminary tests before individual instructions are executed. It checks for a valid opcode (and traps if the opcode is invalid), and then executes the instruction if there is no pending trap. The execute procedure, which defines the behavior of individual instructions, can be found below in parts related to specific instructions.

```

% verify that the opcode is a valid one
disptest(0) :-
    test(inst, op2, 0), !,
    set(p, trap, 1),
    set(p, illegal_instruction, 1).
disptest(0) :-
    !.
disptest(2) :-
    test(inst, op3, unassigned), !,
    set(p, trap, 1),
    set(p, illegal_instruction, 1).
disptest(2) :-
    !.
disptest(3) :-
    test(inst, op3, unassigned), !,
    set(p, trap, 1),
    set(p, illegal_instruction, 1).
disptest(3) :-
    !.
disptest(Op) :-
    !.

```

```

% if there is not a trap pending, execute the instruction
dispatch(Op) :-
    test(p, trap, 0), !,
    execute(Op).
dispatch(Op) :-
    !.

```

## 9.9. Updating the Program Counter

The code to update the PC was defined. It tests for instructions that themselves update the PC, and for those does nothing.

```

%-- Test for PC changing instructions:
%-- CALL, RETT, JMPL, Bicc, FBfcc, CBccc, Ticc do not change PC.
updatePC(call) :-
    !.
updatePC(rett) :-
    !.
updatePC(jmpl) :-
    !.
... (several instructions omitted for brevity)
updatePC(OP) :-
    access(npc, PC),
    set(pc, PC),
    NPC is PC + 4,
    set(npc, NPC).

```

## 9.10. Individual Instructions: Arithmetic, Logical, and Shift

Individual instructions were defined in order of relative complexity, simplest first (not in the order they appear in Appendix C.9). For brevity, only example instructions will be presented here.

First, the basic arithmetic, logical, and shift instructions were defined. These have a characteristic access-access-operate-set structure: the source operands are retrieved from the register file and are used to compute a result, which is then stored into a destination register in the register file. Alternatively, one of the source operands can be an immediate. In Prolog this alternative is defined in a separate clause.

The add instruction is typical of such instructions:

```

execute(add) :-
    test(inst, i, 0), !,
    access(inst, rs1, RS1),
    rval(port(r, r1), RS1, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    access(inst, rd, RD),
    wval(port(r, w), RD, D).
execute(add) :-
    access(inst, simm13, IMM),
    signExtend(IMM, 13, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    access(inst, rd, RD),
    wval(port(r, w), RD, D).

```

A few new operators, not found in Prolog, were defined for these instructions (including addc, subc, and xor).

## 9.11. Individual Instructions: Load and Store

The load and store instructions were specified, including the privileged alternate space -A variants. The definition of these instructions was preceded by the construction of several common subroutines. The load and store word instructions use these subroutines:

```
execute(ld) :-  
    setLoadAddress,  
    setAddrSpace,  
    testWordAlignment,  
    readData,  
    testAccess,  
    loadDataWord.
```

```
execute(st) :-  
    setStoreAddress,  
    setAddrSpace,  
    testWordAlignment,  
    storeDataWord,  
    storeData,  
    testAccess.
```

The setLoadAddress and setStoreAddress routines compute the respective effective load or store address and set the memory address register accordingly. Both compute the effective address using either two register operands (address and offset) or a register and an immediate. The setStoreAddress routine also checks the trap bit and, if set, does not store the address.

```
setLoadAddress :-  
    % two registers  
    test(inst, i, 0), !,  
    access(inst, rs1, I),  
    rval(port(r, r1), I, RI),  
    access(inst, rs2, J),  
    rval(port(r, r2), J, RJ),  
    Address is RI + RJ,  
    set(memAR, Address).
```

```
setLoadAddress :-  
    % register plus immediate  
    access(inst, rs1, I),  
    rval(port(r, r1), I, RI),  
    access(inst, simm13, J),  
    signExtend(J, 13, SignedJ),  
    Address is RI + SignedJ,  
    set(memAR, Address).
```

```
setStoreAddress :-  
    test(p, trap, 1), !.
```

```
setStoreAddress :-  
    % two registers  
    test(inst, i, 0), !,  
    access(inst, rs1, I),  
    rval(port(r, r1), I, RI),  
    access(inst, rs2, J),  
    rval(port(r, r2), J, RJ),  
    Address is RI + RJ,  
    set(memAR, Address).
```

```
setStoreAddress :-  
    % register plus immediate  
    access(inst, rs1, I),  
    rval(port(r, r1), I, RI),  
    access(inst, simm13, J),
```

```

    signExtend(J, 13, SignedJ),
    Address is RI + SignedJ,
    set(memAR, Address).

```

Depending on the supervisor mode bit in the psr, the setAddrSpace routine sets the memory address space register.

```

setAddrSpace :-
    test(psr, s, 0), !,
    set(memAS, 10).
setAddrSpace :-
    set(memAS, 11).

```

The testWordAlignment routine tests for word alignment errors.

```

testWordAlignment :-
    test(memAR, wordAddr, 0), !.
testWordAlignment :-
    set(p, trap, 1),
    set(p, mem_addr_not_aligned, 1).

```

The readData routine performs the actual memory read.

```

readData :-
    test(p, trap, 0), !,
    mem_read.
readData :- !.

```

The testAccess routine tests the memory access error register (one bit) for memory access errors, and sets the processor state accordingly.

```

testAccess :-
    test(memAE, 1), !,
    set(p, trap, 1),
    set(p, data_access_exception, 1).
testAccess :- !.

```

The loadDataWord routine moves a word of data from the memory data register into the register file. This movement is not performed if the trap bit is set or if the destination register is zero (which contains a read-only constant zero).

```

loadDataWord :-
    test(p, trap, 1), !.
loadDataWord :-
    test(inst, rd, 0), !.
loadDataWord :-
    access(memDR, Data),
    access(inst, rd, RD),
    wval(port(r, w), RD, Data).

```

The storeDataWord routine is the store analog of loadDataWord -- it moves a word of data from the register file into the memory data register, checking the trap bit.

```

storeDataWord :-
    test(p, trap, 1), !.
storeDataWord :-
    set(memBM, 15),
    access(inst, rd, RD),
    rval(port(r, r1), RD, Data),
    set(memDR, Data).

```

The storeData routine performs the actual memory write.

```

storeData :-
    test(p, trap, 0), !,

```

```
mem_write.  
storeData :- !.
```

## 9.12. Individual Instructions: Call and Jump and Link

The call and jump and link instructions were defined. These instructions are significant because they change the program counter. The call instruction saves the program counter and jumps to a long immediate address. The jump and link instruction also saves the program counter and jumps to a location computed either from two register sources or a register and an immediate, checking the alignment of the computed location.

```
execute(call) :-  
    access(pc, SavedPC),  
    wval(port(r, w), 15, SavedPC),  
    access(npc, ThisPC),  
    set(pc, ThisPC),  
    access(inst, disp30, Disp),  
    AlignedDisp is Disp /align30,  
    NextPC is ThisPC + AlignedDisp,  
    set(npc, NextPC).
```

```
execute(jmpl) :-  
    test(inst, i, 0), !,  
    access(inst, rs1, RS1),  
    rval(port(r, r1), RS1, S1),  
    access(inst, rs2, RS2),  
    rval(port(r, r2), RS2, S2),  
    Address is S1 + S2,  
    set(tempAddr, Address),  
    jmpJump.
```

```
execute(jmpl) :-  
    access(inst, simm13, IMM),  
    signExtend(IMM, 13, S1),  
    access(inst, rs2, RS2),  
    rval(port(r, r2), RS2, S2),  
    Address is S1 + S2,  
    set(tempAddr, Address),  
    jmpJump.
```

```
jmpJump :-  
    test(tempAddr, byteAddr, 0), !,  
    access(pc, SavedPC),  
    access(inst, rd, RD),  
    wval(port(r, w), RD, SavedPC),  
    access(npc, ThisPC),  
    set(pc, ThisPC),  
    access(tempAddr, NextPC),  
    set(npc, NextPC).
```

```
jmpJump :-  
    set(p, trap, 1),  
    set(p, mem_address_not_aligned, 1).
```

## 9.13. Individual Instructions: Save and Restore

The save and restore instructions were defined. These instructions are somewhat subtle, since they change the register window.

```
execute(save) :-  
    saveMakeCWP,  
    maskCWP,  
    saveTestCWP, !.
```

```
execute(restore) :-
    restoreMakeCWP,
    maskCWP,
    restoreTestCWP, !.
```

The saveMakeCWP routine decrements the current window pointer, using the cwp field from the psr register. The routine, in performing modular addition, assumes the number of register windows to be four.

```
saveMakeCWP :-
    test(psr, cwp, 0), !,
    % modulo subtraction -- NewCWP is NWindows - 1
    % nwindows = 4
    set(tempCWP, 3).
saveMakeCWP :-
    access(psr, cwp, OldCWP),
    NewCWP is OldCWP - 1,
    set(tempCWP, NewCWP).
```

The maskCWP routine uses the current window pointer to produce a mask in the tempMask register, with a one in the bit position of the currently active window. This mask is then tested for validity against the permanent mask contained in the window invalid mask register. Windowed register files were developed by a small team of graduate students at U.C. Berkeley who had little hardware experience.

```
maskCWP :-
    set(tempMask, 1),
    access(tempMask, BaseMask),
    access(tempCWP, CWP),
    access(wim, WIM),
    ResultMask is (BaseMask << CWP) ^ WIM,
    set(tempMask, ResultMask).
```

The saveTestCWP routine tests the result of the maskCWP routine, and sets the window overflow flag if appropriate.

```
saveTestCWP :-
    test(tempMask, 0), !,
    setCWP.
saveTestCWP :-
    set(p, trap, 1),
    set(p, window_overflow, 1).
```

The restoreMakeCWP routine is the restore analog of saveMakeCWP. It increments the current window pointer.

```
restoreMakeCWP :-
    test(psr, cwp, 3), !,
    % nwindows = 4
    set(tempCWP, 0).
restoreMakeCWP :-
    access(psr, cwp, OldCWP),
    NewCWP is OldCWP + 1,
    set(tempCWP, NewCWP).
```

The restoreTestCWP is the restore version of saveTestCWP, setting the underflow flag.

```
restoreTestCWP :-
    test(tempMask, 0), !,
    setCWP.
restoreTestCWP :-
    set(p, trap, 1),
    set(p, window_underflow, 1).
```

The setCWP routine sets the current window pointer of the psr register and performs an add instruction (with

the source operands from the old window and the destination in the new one).

```
setCWP :-
    test(inst, i, 0), !,
    access(inst, rs1, RS1),
    rval(port(r, r1), RS1, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    access(tempCWP, CWP),
    set(psr, cwp, CWP),
    access(inst, rd, RD),
    wval(port(r, w), RD, D).
```

```
setCWP :-
    access(inst, simm13, IMM),
    signExtend(IMM, 13, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    access(tempCWP, CWP),
    set(psr, cwp, CWP),
    access(inst, rd, RD),
    wval(port(r, w), RD, D).
```

#### 9.14. Individual Instructions: Return from Trap

The return from trap instruction was defined. It changes the register window, and does considerable error checking.

```
execute(rett) :-
    rettCWP,
    rettAddress,
    maskCWP,
    rettTest.
```

The rettCWP routine is analogous to restoreMakeCWP, incrementing the current window pointer.

```
rettCWP :-
    test(psr, cwp, 3), !,
    % modulo addition -- NewCWP is NWindows + 1
    % nwindows = 4
    set(tempCWP, 0).
rettCWP :-
    access(psr, cwp, OldCWP),
    NewCWP is OldCWP + 1,
    set(tempCWP, NewCWP).
```

The rettAddress routine computes the target address and stores the result in the tempAddr register, pending the outcome of the rettTest routine.

```
rettAddress :-
    test(inst, i, 0), !,
    access(inst, rs1, RS1),
    rval(port(r, r1), RS1, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    Address is S1 + S2,
    set(tempAddr, Address).
rettAddress :-
    access(inst, simm13, IMM),
    signExtend(IMM, 13, S1),
    access(inst, rs2, RS2),
```

```

    rval(port(r, r2), RS2, S2),
    Address is S1 + S2,
    set(tempAddr, Address).

```

The rettTest and rettET routines test for various conditions: traps enabled, user mode, invalid window, and address not aligned. If these conditions obtain various error status bits are set; if they do not a return is performed (changing the pc and the cwp).

```

% ET = 1 (traps enabled)
rettTest :-
    test(psr, et, 1), !,
    set(p, trap, 1),
    rettET.
% S = 0 (not supervisor mode)
rettTest :-
    test(psr, s, 0), !,
    set(p, trap, 1),
    set(p, privileged_instruction, 1),
    set(p, execute_mode, 0),
    set(p, error_mode, 1),
    set(tbr, tt, tmask5).
% window invalid
rettTest :-
    test(tempMask, 0), !,
    set(p, trap, 1),
    set(p, window_underflow, 1),
    set(p, execute_mode, 0),
    set(p, error_mode, 1),
    set(tbr, tt, tmask11).
% OK
rettTest :-
    test(tempAddr, byteAddr, 0), !,
    set(psr, et, 1),
    access(npc, ThisPC),
    set(pc, ThisPC),
    access(tempAddr, NextPC),
    set(npc, NextPC),
    access(tempCWP, CWP),
    set(psr, cwp, CWP),
    access(psr, ps, S),
    set(psr, s, S).
% address not aligned
rettTest :-
    set(p, trap, 1),
    set(p, mem_address_not_aligned, 1),
    set(p, execute_mode, 0),
    set(p, error_mode, 1),
    set(tbr, tt, tmask12).

rettET :-
    test(psr, s, 0), !,
    set(p, privileged_instruction, 1).
rettET :-
    set(p, illegal_instruction, 1).

```

## 9.15. Individual Instructions: Branch and Trap on Condition Codes

The branch and trap on condition instructions were defined. Each performs a test and then, based on the result, either changes the pc or continues linear execution. The representative branch if not zero and branch if zero instructions, with branch and nobranch routines (note the handling of annulled instructions in nobranch):

```

execute(bne) :-
    test(psr, z, 0), !,
    branch.
execute(bne) :-
    nobranch.

execute(be) :-
    test(psr, z, 1), !,
    branch.
execute(be) :-
    nobranch.

branch :-
    access(npc, ThisPC),
    set(pc, ThisPC),
    access(inst, disp22, Disp),
    AlignedDisp is Disp /align22,
    % align22 is 22 bit mask, 2 low bits zero
    signExtend(AlignedDisp, 22, Offset),
    NextPC is ThisPC + Offset,
    set(npc, NextPC).

nobranch :-
    test(inst, a, 0), !,
    access(npc, ThisPC),
    set(pc, ThisPC),
    NextPC is ThisPC + 4,
    set(npc, NextPC).
nobranch :-
    access(npc, ThisPC),
    set(pc, ThisPC),
    NextPC is ThisPC + 4,
    set(npc, NextPC),
    set(p, annul, 1).

```

## 10. Appendix: Improving the Specification

### 10.1. Trap Handling: Dispatch

A SPARC designer was consulted with respect to the actual implementation of the trap handling hardware.

Trap dispatch uses a sequence of tests to determine which trap should be activated. It is specified thus:

```

trapsel :-
    test(p, data_store_error, 1), !,
    set(tbr, tt, tmask1).
trapsel :-
    test(p, instruction_access_error, 1), !,
    set(tbr, tt, tmask2).
trapsel :-
    test(p, r_register_access_error, 1), !,
    set(tbr, tt, tmask3).
...

```

There are 23 arms in this conditional, all tested sequentially. In a real processor this test is performed by a single PLA that does the priority encoded testing in a single operation.

A priority encoder construct was developed that would support such an implementation. The various conditions to be tested are aggregated into a named list, which can then be used for dispatching. For example,

```

statePriority(
    [ data_store_error,
      instruction_access_error,

```

```

        r_register_access_error,
        ...],
        Priority),

```

defines the trap conditional list, and

```

        trapset(Priority),

```

invokes the trap selection procedure; the individual conditional clauses appear as

```

        trap(data_store_error) :-
            set(tbr, tt, tmask1), !.
        trap(instruction_access_error) :-
            set(tbr, tt, tmask2), !.
        trap(r_register_access_error) :-
            set(tbr, tt, tmask3), !.
        ...

```

This form requires that all conditions tested evaluate to 1.

## 10.2. Register Windows

The management of register windows requires modular arithmetic (modulo the number of register windows). In the plain specification this is done via a test. Consider, for example, the changing of the window pointer in the save instruction:

```

        saveCWP :-
            test(psr, cwp, 0), !,
            % nwindows = 4
            set(tempCWP, 3).
        saveCWP :-
            access(psr, cwp, OldCWP),
            NewCWP is OldCWP - 1,
            set(tempCWP, NewCWP).

```

This calculation requires an extra cycle for the test; a counter of the right size avoids this test.

Register windows also involve a one's hot window invalid mask, which indicates that certain windows cannot be used. The window pointer must be tested against this mask, which requires that the pointer, a numeric index, must be converted to one's hot form. The Prolog code to do this is:

```

        maskCWP :-
            set(tempMask, 1),
            access(tempMask, BaseMask),
            access(tempCWP, CWP),
            access(wim, WIM),
            ResultMask is (BaseMask << CWP) /WIM,
            set(tempMask, ResultMask).

```

This can be done more simply with a PLA in the library.

## 10.3. Operands: Conditional Sources and Destinations

Consider the add instruction:

```

        execute(add) :-
            test(inst, i, 0), !,
            access(inst, rs1, RS1),
            rval(port(r, r1), RS1, S1),
            access(inst, rs2, RS2),
            rval(port(r, r2), RS2, S2),
            D is S1 + S2,
            resultWrite(D).
        execute(add) :-
            access(inst, simm13, IMM),

```

```

    signExtend(IMM, 13, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    resultWrite(D).
resultWrite(D) :-
    test(inst, rd, 0), !.
resultWrite(D) :-
    access(inst, rd, RD),
    wval(port(r, w), RD, D).

```

Note the two tests here. One determines if the instruction uses an immediate first argument instead of a register. The other determines if the destination register is zero (in which case the write is disabled). In a real processor these tests are not performed sequentially, but in parallel, with the test lines enabling different logic.

In the Prolog context the result write test can be built into the register file handling library routines (wval here). The immediate test is harder to optimize. One possibility is to make the compilation system smarter and move the test into the caller.

## 10.4. Sign Extension

The preparation of immediate and jump displacement operands requires sign extension. In Prolog this is specified with tests and masks, for example:

```

signExtend8(Byte, Word) :-
    Byte > 127, !,
    Word is Byte / mmask8.
signExtend8(Word, Word) :- !.

```

This test costs another cycle. It can be optimized through implementation as a library function.

## 10.5. Condition Codes

Various arithmetic and logical instructions, such as addcc and orcc, set condition codes. These condition codes are functions of various result and operand bits. Consider the zero condition code and the add instruction:

```

setAddZ :-
    test(result, 0), !,
    set(psr, z, 1).
setAddZ :-
    set(psr, z, 0).

```

Once again, this testing is expensive. This should be implemented as logic, the result of which (the zero test) is routed to the psr register. This routing is done in Prolog with the tval procedure, which routes data path tests (usually used for control) into data path elements. The addcc instruction, for example (the register-immediate version), is

```

execute(addcc) :-
    access(inst, simm13, IMM),
    signExtend(IMM, 13, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    access(inst, rd, RD),
    wval(port(r, w), RD, D),
    setFlagNeg(D),
    setFlagZero(D),
    setFlagOverflow(D),
    setFlagCarry(D).

```

with

```

setFlagNeg(Data) :-
    tval(isNeg, Data, Bool),

```

```

    set(psr, n, Bool).
setFlagZero(Data) :-
    tval(isZero, Data, Bool),
    set(psr, z, Bool).
setFlagOverflow(Data) :-
    tval(isOverflow, Data, Bool),
    set(psr, v, Bool).
setFlagCarry(Data) :-
    tval(isCarry, Data, Bool),
    set(psr, c, Bool).

```

setting the condition codes. Each of the is... values is the output of a library function that produces the proper value (isZero, for example, is the output of the zero test).

Note that most of the above fixes involve adding functions to the library. This is problematic for two reasons. First, modifying the library is not entirely straightforward. Second, it hides more and more of the design, moving it from the specification into magical ancillary files.

## 11. Appendix: Translating Prolog into VHDL

### 11.1. Registers: I

The first attempt replaced the Prolog access and set primitives with VHDL equivalents.

In the subset of Prolog processed by Viper, registers and fields are declared via facts, which the access and set procedures reference. For example, the processor state register and its fields are declared with

```

% Processor State Register
stateRegister(psr, 32).
stateField(psr, impl, (31-28)).    % implementation id
stateField(psr, ver, (27-24)).    % implementation version
stateField(psr, icc, (23-20)).    % IU condition codes
    stateField(psr, n, 23).        % negative
    stateField(psr, z, 22).        % zero
    stateField(psr, v, 21).        % overflow
    stateField(psr, c, 20).        % carry
stateField(psr, reserved_PSR, (19-14)). % unused
stateField(psr, ec, 13).          % enable CP
stateField(psr, ef, 12).          % enable FP
stateField(psr, pil, (11-8)).     % acceptable interrupt level
stateField(psr, s, 7).            % supervisor mode
stateField(psr, ps, 6).           % previous s value
stateField(psr, et, 5).           % enable traps
stateField(psr, cwp, (4-0)).      % current window pointer

```

and incrementing the cwp field is done with the code

```

access(psr, cwp, OldCWP),
NewCWP is OldCWP + 1,
set(psr, cwp, NewCWP)...

```

The access routine, using the above definitions, fetches the appropriate bits from the register data word, and the set routine modifies them. In this way the system explicitly maintains the state of all registers and their fields. The user never accesses the contents of a register directly.

This mechanism was copied in VHDL.

A register type was defined:

```

subtype storage is BIT_VECTOR(wordsize - 1 downto 0);

```

The register psr was declared:

```

variable psr : storage;

```

The field names were declared in an enumerated type:

```
type psrField is
  (impl,      -- implementation id
  ver,        -- implementation version
  icc,        -- IU condition codes
  n,          -- negative
  z,          -- zero
  v,          -- overflow
  c,          -- carry
  reserved_PSR, -- unused
  ec,         -- enable CP
  ef,         -- enable FP
  pil,        -- acceptable interrupt level
  s,          -- supervisor mode
  ps,         -- previous s value
  et,         -- enable traps
  cwp);       -- current window pointer
```

and then specific access and set procedures for the psr were defined:

```
procedure accessPSR(variable FieldName : in psrField;
  variable Data : out BIT_VECTOR) is
begin
  -- psr is global
  case FieldName is
    when impl =>
      Data := psr(31 downto 28);
    when ver =>
      Data := psr(27 downto 24);
    ...
    when et =>
      Data := psr(5);
    when cwp =>
      Data := psr(4 downto 0);
  end case;
end accessPSR;

procedure setPSR(variable FieldName : in psrField;
  variable Data : in BIT_VECTOR) is
begin
  -- psr is global
  case Field is
    when impl =>
      psr(31 downto 28) := Data;
    when ver =>
      psr(27 downto 24) := Data;
    ...
    when et =>
      psr(5) := Data;
    when cwp =>
      psr(4 downto 0) := Data;
  end case;
end setPSR;
```

This mechanism is notably "unnatural" -- tedious and rather clumsy, and this detail must be defined by the \*user\*. In Prolog, in contrast, given its built-in data base and pattern matching capabilities, these register specific enumerated types and cases are unnecessary.

## 11.2. Registers: II

A second attempt was made using VHDL records.

Register and register field subtypes were declared:

```
subtype storage is BIT_VECTOR(wordsize - 1 downto 0);
subtype field1 is BIT_VECTOR(0 downto 0); -- for consistency
subtype field2 is BIT_VECTOR(1 downto 0);
subtype field3 is BIT_VECTOR(2 downto 0);
subtype field4 is BIT_VECTOR(3 downto 0);
subtype field5 is BIT_VECTOR(4 downto 0);
subtype field6 is BIT_VECTOR(5 downto 0);
subtype field7 is BIT_VECTOR(6 downto 0);
subtype field8 is BIT_VECTOR(7 downto 0);
...
```

The psr was then declared:

```
type psrRegister is
  record
    impl : field4;
    ver : field4;
    icc : psrRegisterCC;
    reserved_psr : field6;
    ec : field1;
    ef : field1;
    pil : field4;
    s : field1;
    ps : field1;
    et : field1;
    cwp : field5;
  end record;

type psrRegisterCC is
  record
    n : field1;
    z : field1;
    v : field1;
    c : field1;
  end record;

variable psr : psrRegister;
```

This is a relatively straightforward specification. In addition, set and access are not needed -- fields can be referenced directly.

This design has two fatal flaws, however.

First, note that psr is of type psrRegister, not type storage. Every register with different fields is a different type, and cannot be assigned to another type of register. The contents of the memory data register, for example, cannot be assigned to a register in the register file.

Second, several registers have overlapping fields, including the memory address register, the memory data register, and the instruction register (in which various opcode and operand fields overlap). Records do not support this notion.

## 11.3. Registers: III

A third attempt was made using VHDL alias statements.

The VHDL alias statement was designed to solve the second flaw above. It allows a subset of bits in a variable to be given an separate name and referenced independently.

Register and field subtypes were declared as above (in 4). The psr was then defined:

```
variable psr : storage;

alias psr_impl : field4 is psr(31-28); -- id
alias psr_ver : field4 is psr(27-24); -- version
alias psr_icc : field4 is psr(23-20); -- IU condition codes
  alias psr_n : field1 is psr(23); -- negative
  alias psr_z : field1 is psr(22); -- zero
  alias psr_v : field1 is psr(21); -- overflow
  alias psr_c : field1 is psr(20); -- carry
alias reserved_psr : field6 is psr(19-14); -- unused
alias psr_ec : field1 is psr(13); -- enable CP
alias psr_ef : field1 is psr(12); -- enable FP
alias psr_pil : field4 is psr(11-8); -- acceptable int level
alias psr_s : field1 is psr(7); -- supervisor mode
alias psr_ps : field1 is psr(6); -- previous s value
alias psr_et : field1 is psr(5); -- enable traps
alias psr_cwp : field5 is psr(4-0); -- current window ptr
```

Note the naming convention: a field F of register R is always named R\_F.

This allows registers and fields to be freely assignable.

All of the register and field definitions for the SPARC processor (Appendices C.3 and C.4) were defined in this style.

## 11.4. Interfaces

Off chip interfaces were translated from stateInterface declarations in Prolog to VHDL signal declarations, to wit:

```
stateInterface(bp_IRL, (3-0)).
stateInterface(bp_reset_in, 0).
stateInterface(pb_error, 0).
stateInterface(pb_block_ldst_word, 0).
stateInterface(pb_block_ldst_byte, 0).
stateInterface(bp_FPU_present, 0).
stateInterface(bp_FPU_exception, 0).
stateInterface(bp_FPU_cc, (1-0)).
stateInterface(bp_CP_present, 0).
stateInterface(bp_CP_exception, 0).
stateInterface(bp_CP_cc, (1-0)).

signal bp_IRL : field4;
signal bp_reset_in : field1;
signal pb_error : field1;
signal pb_block_ldst_word : field1;
signal pb_block_ldst_byte : field1;
signal bp_FPU_present : field1;
signal bp_FPU_exception : field1;
signal bp_FPU_cc : field2;
signal bp_CP_present : field1;
signal bp_CP_exception : field1;
signal bp_CP_cc : field2;
```

## 11.5. Register Files

Register files were translated into VHDL arrays:

```
type register_file is array(natural range <>) of storage;
```

```

variable r : register_file(0 to (16 * nwindows) - 1);
variable g : register_file(1 to 7);
variable asr : register_file(1 to 31);

```

## 11.6. The add Instruction: I

Given the VHDL alias version of the SPARC register definitions, it was decided to start the translation of the rest of the specification with the add instruction, which would expose many issues without introducing too much complexity.

The first attempt at the add instruction copied the Prolog version as much as possible. The Prolog form for the register to register add (without an immediate operand):

```

execute(add) :-
    access(inst, rs1, RS1),
    rval(port(r, r1), RS1, S1),
    access(inst, rs2, RS2),
    rval(port(r, r2), RS2, S2),
    D is S1 + S2,
    access(inst, rd, RD),
    % test for rd = 0 done in wval
    wval(port(r, w), RD, D).

```

This translates straightforwardly into VHDL:

```

when add =>
    access(inst_rs1, RS1);
    rval(r, r1, RS1, S1);
    access(inst_rs2, RS2);
    rval(r, r2, RS2, S2);
    add(S1, S2, D);
    access(inst_rd, RD);
    wval(r, w, RD, D);

```

Additional declarations and a library of support procedures are required, however.

First, access and set must be defined; they are trivial. These routines are in the library and are not synthesized.

```

procedure access(field: in BIT_VECTOR; data: out BIT_VECTOR) is
begin
    data := field;
end access;

procedure set(field: out BIT_VECTOR; data: in BIT_VECTOR) is
begin
    field := data;
end set;

```

Note the slight difference with the 3 argument Prolog versions; the VHDL aliases have the register name built into the field name.

The add procedure must also be defined. Unlike Prolog, which is untyped, VHDL requires conversion between bits (BIT\_VECTORS) and integers. Hence the add procedure:

```

procedure add(S1: in BIT_VECTOR;
             S2: in BIT_VECTOR;
             D: out BIT_VECTOR) is
    variable IS1 : INTEGER;
    variable IS2 : INTEGER;
    variable ID  : INTEGER;
    -- bits_to_int and int_to_bits in VHDL cookbook, page 7-8
begin
    IS1 := bits_to_int(S1);

```

```

    IS2 := bits_to_int(S2);
    ID := IS1 + IS2;
    D := int_to_bits(ID);
end add;

```

This procedure is in the library and is not synthesized.

Next, the register file routines `rval` and `wval` must be defined. These routines are in the library and are not synthesized.

```

procedure rval(regfile: in register_file;
               regport: in port_name;
               index: in BIT_VECTOR;
               data: out BIT_VECTOR) is
-- regport is used in synthesis for scheduling
  variable I : INTEGER;
begin
  I := bits_to_int(index);
  data := regfile(I);
end rval;

procedure wval(regfile: inout register_file;
               regport: in port_name;
               index: in BIT_VECTOR;
               data: in BIT_VECTOR) is
-- regfile is inout so that the other registers in the file
-- are passed through.
  variable I : INTEGER;
begin
  I := bits_to_int(index);
  regfile(I) := data;
end wval;

```

Here, as with `add`, conversion between bits and ints is needed. Also, a type declaration for `port_name`, the list of possible read and write ports on the register file, is required.

```

type port_name is (r1, r2, w);

```

Finally, the variables `S1`, `S2`, `D`, `RS1`, `RS2`, and `RD` must be declared (in a block or environment global to the instruction dispatch case statement). They can be declared as bit vectors, but it is useful and descriptive to declare two bit vector subtypes of which these variables are instances:

```

-- non-storage variables (realized as buses)
-- From VHDL's point of view these are basically the same
-- thing as registers; from the synthesizer's point of
-- view they are very different.
subtype connection is BIT_VECTOR(wordsize - 1 downto 0);

-- size is number of bits in index
-- Other register files (g, asr) are different sizes;
-- their indices are different sizes and they require
-- different subtypes.
subtype register_index
  is BIT_VECTOR(log2(16*nwindows) downto 1);

variable S1, S2, D : connection;
variable RS1, RS2, RD : register_index;

```

## 11.7. The `addcc` Instruction

The `addcc` instruction was then defined in a similar style:

```

when addcc =>
  access(inst_rs1, RS1);
  rval(r, r1, RS1, S1);
  access(inst_rs2, RS2);
  rval(r, r2, RS2, S2);
  add(S1, S2, D, N, Z, V, C);
  access(inst_rd, RD);
  wval(r, w, RD, D);
  set(psr_n, N);
  set(psr_z, Z);
  set(psr_v, V);
  set(psr_c, C);

```

This simply requires a more complex add function (in the library, not synthesized).

```

procedure add(S1: in BIT_VECTOR; -- also connection
             S2: in BIT_VECTOR;
             D: out BIT_VECTOR;
             N: out BIT_VECTOR; -- also field1
             Z: out BIT_VECTOR;
             V: out BIT_VECTOR;
             C: out BIT_VECTOR) is
  variable IS1 : INTEGER;
  variable IS2 : INTEGER;
  variable ID : INTEGER;
  -- bits_to_int and int_to_bits in VHDL cookbook, page 7-8
begin
  IS1 := bits_to_int(S1);
  IS2 := bits_to_int(S2);
  ID := IS1 + IS2;
  D := int_to_bits(ID);
  N := D(31);
  if result = 0 then
    Z := 1;
  else
    Z := 0;
  end if;
  -- see SPARC Appendix C.9, page 173
  V := (S1(31) and S2(31) and (not D(31))) or
        ((not S1(31)) and not S2(31) and D(31));
  C := (S1(31) and S2(31)) or
        ((not D(31)) and (S1(31) or S2(31)));
end add;

```

Also, N, Z, V, and C must be declared with S1, S2, etc.

```
variable N, Z, V, C : field1;
```

## 11.8. The add Instruction: II

A second version of the add instruction was defined, without set, access, rval, and wval, but with connection variables:

```

when add =>
  S1 := r(inst_rs1);
  S2 := r(inst_rs2);
  add(S1, S2, D);
  if inst_rd /= 0 then
    r(inst_rd) := D;
  end if;

```

This is obviously much more straightforward from a human standpoint than the first version, and the synthe-

sizer could fairly easily derive the first version, which it wants, from this, with two exceptions.

The first is the if-then test for the result register. From the synthesizer's point of view this test would be more easily handled if it were encapsulated in a library procedure, a given fragment of logic. Otherwise, for all instructions that perform this test, the synthesizer would have to recognize the test as a common fragment and optimize it.

Given such a procedure, `r_store`, say, the definition would appear

```
when add =>
  S1 := r(inst_rs1);
  S2 := r(inst_rs2);
  add(S1, S2, D);
  r_store(inst_rd, D);
```

The second exception is the loss of register ports, with the loss of `rval` and `wval`. These ports require the user to schedule references to the register file, binding them to specific ports. This scheduling could be done by the synthesizer, but it is not done currently.

There is, however, a larger flaw. Remember that `inst_rs1` is a `BIT_VECTOR`, not an integer as is required for array references. This version is not executable VHDL.

### 11.9. The add Instruction: III

A third version was defined eliminating unnecessary variables and fixing the register indexing problem.

```
when add =>
  add(r(i(inst_rs1)), r(i(inst_rs2)), D);
  r_store(inst_rd, D);
```

The `i` procedure simply does a `bits_to_int` conversion.

The synthesizer could derive the second version from this one.

### 11.10. The add Instruction: IV

A fourth version was defined, which included handling an immediate operand.

```
when add =>
  if inst_i = 0 then
    S2 := r(i(inst_rs2));
  else
    S2 := sign_extend(inst_simm13);
  end if;
  add(r(i(inst_rs1)), S2, D);
  r_store(inst_rd, D);
```

with `sign_extend` the obvious (unsynthesized) library routine.

Here, as above with `r_store`, there is the issue of whether the immediate should be handled completely in the instruction definition, or whether a library routine should be defined.

### 11.11. The add Instruction: V

A final version was defined, using library routines to handle sources and destinations.

```
when add =>
  S1 := rs1;
  S2 := rs2imm13;
  add(S1, S2, D);
  rd(D);

function rs1 return BIT_VECTOR is
  variable I : BIT_VECTOR;
  variable S : BIT_VECTOR;
begin
```

```

    access(inst_rs1, I);
    rval(r, r1, I, S);
    return S;
end rs1;

function rs2imm13 return BIT_VECTOR is
    variable I : BIT_VECTOR;
    variable S : BIT_VECTOR;
begin
    if inst_1 = 0 then
        access(inst_rs2, I);
        rval(r, r2, I, S);
    else
        S := sign_extend(inst_simm13);
    end if;
    return S;
end rs2imm13;

procedure rd(D: in BIT_VECTOR) is
    variable I: BIT_VECTOR;
begin
    access(inst_rd, I);
    -- conversion to integer necessary?
    if I /= 0 then
        wval(r, w, I, D);
    end if;
end rd;

```

This makes the definition of the actual instruction short, and defines procedures that are used extensively in other instructions.

In general, instruction definitions describe data movement through the data path. The philosophy is to put operations that will not be performed by data path functional units (such as tests done by specialized logic rather than ALUs) into the library. The procedures `rs1`, `rs2imm13`, and `rd` above could be put in the library.

## 11.12. Load and Store Instructions

With the experience of the `add` instruction, the load and store instructions were translated, since they are substantially different in form from `add`.

With the register and field definitions done, including those for the memory interface, the load and store instructions were easily defined (and are not very interesting).

The `ld` instruction, for example, in Prolog:

```

execute(ld) :-
    setLoadAddress,
    setAddrSpace,
    testWordAlignment,
    readData,
    testAccess,
    loadDataWord.

```

and in VHDL:

```

when ld =>
    setLoadAddress;
    setAddrSpace;
    testWordAlignment;
    readData;
    testAccess;
    loadDataWord;

```

with two of the supporting procedures, for example:

```
procedure setLoadAddress is
begin
  S2 := rs2_or_imm13(inst_i, inst_rs2, inst_simm13);
  add(r(i(inst_rs1)), S2, D);
  memAR := D;
end setLoadAddress;

procedure setAddrSpace is
begin
  if psr_s = 0 then
    memAS := 10;
  else memAS := 11;
  end if;
end setAddrSpace;
```

### 11.13. The Main Loop

The main control loop of the SPARC processor (Appendix C.5) was then translated from Prolog into VHDL. The translation was relatively straightforward, consisting primarily of status tests and bit assignments.

Central to this translation was the definition of various internal status bits, which had been defined in the Prolog version and had been translated into VHDL along with the register definitions (see above).

```
variable p : storage;

alias p_annul : field1 is p(32);
alias p_cp_disabled : field1 is p(31);
alias p_cp_exception : field1 is p(30);
alias p_data_access_error : field1 is p(29);
alias p_data_access_exception : field1 is p(28);
alias p_data_store_error : field1 is p(27);
alias p_division_by_zero : field1 is p(26);
alias p_error_mode : field1 is p(25);
alias p_execute_mode : field1 is p(24);
alias p_fp_disabled : field1 is p(23);
alias p_fp_exception : field1 is p(22);
alias p_illegal_instruction : field1 is p(21);
alias p_instruction_access_error : field1 is p(20);
alias p_instruction_address_exception : field1 is p(19);
alias p_mem_address_not_aligned : field1 is p(18);
alias p_privileged_instruction : field1 is p(17);
alias p_r_register_access_error : field1 is p(16);
alias p_reset_mode : field1 is p(15);
alias p_reset_trap : field1 is p(14);
alias p_tag_overflow : field1 is p(13);
alias p_trap : field1 is p(12);
alias p_trap_instruction : field1 is p(11);
alias p_unimplemented_FLUSH : field1 is p(10);
alias p_window_overflow : field1 is p(9);
alias p_window_underflow : field1 is p(8);
```

Given these definitions...

```
-- main run loop
loop
  reset;
  error;
  execute;
end loop;
```

```

-- reset
procedure reset is
begin
  while bp_reset_in = 1 loop
    p_reset_mode := 1;
  end loop;
  if p_reset_mode = 1 then
    p_reset_mode := 0;
    p_execute_mode := 1;
    p_trap := 1;
    p_reset_trap := 1;
  end if;
end reset;

-- error
procedure error is
begin
  if p_error_mode = 1 then
    while bp_reset_in = 0 loop
      null;
      -- control synthesis must handle null
    end loop;
    p_error_mode := 0;
    p_reset_mode := 1;
    pb_error <= 0;
  end if;
end error;

-- execute
procedure execute is
begin
  trap;
  if p_execute_mode = 1 then
    fetch;
    -- bad address
    if ((memAE = 1) and (annul = 0)) then
      p_trap := 1;
      p_instruction_address_exception := 1;
      return;
      -- control synthesis must handle return
    end if;
    -- annulled
    if p_annul = 1 then
      p_annul := 0;
      pc := npc;
      npc := npc + 4;
      return;
    end if;
    -- normal
    execute_dispatch;
    updatePC;
  end execute;

```

... and so on.

This style is substantially more readable than the equivalent Prolog which, using clause selection in place of if-thens, is more fragmented. The error procedure, for example, appears in Prolog:

```

% loop while error is high
error :-

```

```

    test(p, error_mode, 1), !,
    errorReset,
    error.
error :- !.

errorReset :-
    test(bp_reset_in, 1), !,
    % initialize state
    set(p, error_mode, 0),
    set(p, reset_mode, 1),
    set(pb_error, 0).
errorReset :- !.

```

## 11.14. Instruction Dispatch and Traps

Appendix C.6, Instruction Dispatch, was translated. This is a large case statement, otherwise uninteresting.

Appendix C.8, Traps, was translated. Again, because Prolog does not have if-thens, the Prolog specification is fragmented. In contrast, the VHDL appears less spread out and more readable and coherent. In the code below, each commented segment is a separate procedure in Prolog, each consisting of at least two clauses.

```

procedure execute_trap is
begin
    trapsel;      -- a large case statement
    trapex;      -- set 16 assorted status flags to 0
    -- trappc
    if p_error_mode = 0 then
        psr_et := 0;
        psr_ps := psr_s;
    -- trapCWP (use library function)
        window_decrement(psr_cwp);
    -- trappcLast
        if p_annul = 0 then
            r(17) := pc;
            r(18) := npc;
        else
            r(17) := npc;
            r(18) := npc + 4;
            p_annul := 0;
        end if;
        psr_s := 1;
    -- trappcNext
        if p_reset_trap = 0 then
            pc := tbr;
            npc := tbr + 4;
        else
            pc := 0;
            npc := 4;
            p_reset_trap := 0;
        end if;
    end if;
end execute_trap;

```