THE HIGH-LEVEL SYNTHESIS OF MICROPROCESSORS USING INSTRUCTION FREQUENCY STATISTICS

by

William Read Bush

Memorandum No. UCB/ERL M92/109

15 May 1992

COVER PAGE

THE HIGH-LEVEL SYNTHESIS OF MICROPROCESSORS USING INSTRUCTION FREQUENCY STATISTICS

by

William Read Bush

Memorandum No. UCB/ERL M92/109

15 May 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

THE HIGH-LEVEL SYNTHESIS OF MICROPROCESSORS USING INSTRUCTION FREQUENCY STATISTICS

by

William Read Bush

Memorandum No. UCB/ERL M92/109

15 May 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The High-Level Synthesis of Microprocessors Using Instruction Frequency Statistics

William R. Bush

May 1992

Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

Abstract

The RISC approach to computer design optimizes commonly executed instructions. This work automates that process at the microarchitectural level, synthesizing optimized microprocessor implementations from behavioral, program-like specifications. It uses instruction frequency statistics to guide optimization, to improve both the cost-performance ratio and the absolute performance of synthesized designs. Two specific optimization techniques have been investigated. The first is trace scheduling, which was developed for microcode and very long instruction word compilers, and was adapted in this work to the hardware synthesis domain. The second is a new technique for allocating hardware resources based on dividing hardware operations into two categories, those required to implement a design, and those that optionally improve its performance; optional operations are allocated in decreasing order of importance.

A hardware synthesis system was constructed to test these techniques. Various versions of two substantial microprocessors, the 6502 and the BAM (a RISC processor extended to support Prolog), were synthesized in a series of experiments. In general, trace scheduling increased the throughput of synthesized hardware from 10% to 34%, at the cost of proportionally larger circuit area. The new allocation technique demonstrated limited applicability in microprocessor optimization, but did make possible the automatic generation of specialized microprocessors for applications that only use a subset of instructions.

The High-Level Synthesis of Microprocessors

Using Instruction Frequency Statistics

by

William Read Bush

The High-Level Synthesis of Microprocessors Using Instruction Frequency Statistics

By

William Read Bush

A. B. (Harvard University) 1972
J. D. (Boston University School of Law) 1977
M. S. (University of California at Berkeley) 1985

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved:

Co-Chair: [illegible] 12/23/91
Date

Co-Chair: R.K. Brayton 12/20/91

Randy H. Katz 12/18/91

[illegible] 12/18/91

* * * * * * * * * * * * * * * * * * * *

Acknowledgements

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD): Arpa Order No. 6501, monitored by the Office of Naval Research under Contract No. N00014-88-K-0579, and Arpa Order No. 4871, monitored by the Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

I would like to thank my research advisor, Al Despain, for timely insights, for encouragement, and for creating the research environment of the Aquarius project. I want to thank my co-chair, Bob Brayton, for taking on co-chair duties, and committee members Randy Katz and Peter Berck for experiencing both my qualifying exam and dissertation.

I want to thank Joan Pendleton, of Harvest VLSI, for the use of her cell library, and for the generous and extended use of computing resources. Frank Spies, of Harvest VLSI, provided the SPEC benchmark statistics. Steve Schoettler, of UC Berkeley, wrote the initial 6502 specification in Prolog.

I would like to thank Kathryn Crabtree and Rene Smith, of UC Berkeley, for invaluable help in navigating the extensive Berkeley bureaucratic maze, and fellow students Rick McGeer, Dave Ungar, and Dain Samples, for timely advice and aid.

Most importantly, I wish to thank my parents, Walter and Sally, and my comrade-in-arms, Ellen Peel, for their enduring support.

The High-Level Synthesis of Microprocessors

Using Instruction Frequency Statistics

Table of Contents

List of Tables and Figures

Chapter One: Introduction

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.

This dissertation 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 ([SPEC]).

The basic question addressed by this work is, *how effectively can the quality of automatically generated microprocessors be improved through the use of instruction frequency statistics?* Quality in this context means either absolute performance (making a microprocessor absolutely faster than it would otherwise be), or the ratio of performance to cost (making a more cost effective microprocessor). Since the hardware synthesis system developed to answer this question can synthesize more than microprocessors, the work also touches the more general question, *how effectively can high-level hardware synthesis be improved through the application of frequency of use information?*

Posing these questions immediately raises subquestions.

- How are microprocessors generated automatically?
- How can their quality be improved by using instruction frequencies?
- How can such improvements be measured?
- More generally, what is high-level hardware synthesis and how is it performed?

1. How are microprocessors generated automatically?

In this work, automatically generating a microprocessor means producing a microprocessor implementation from a specification automatically using a design automation computer program.

1.1. Specifications and Implementations

A complete microprocessor design exists at several levels of detail ([Arch-Hayes], [Arch-H&P], [Arch-Tanenbaum]). At the highest level is the instruction set architecture (ISA), "the portion of machine visible to the programmer or compiler writer" ([Arch-H&P], page 89), that is, to the processor's users. At the lowest level is the actual chip. In between are various levels, each implementing the one above it. The register transfer level (RTL), using functional unit components such as adders, buses, and latches, implements the ISA; the logic gate level implements functional units; the transistor level implements gates; the layout level implements transistors; and the physical chip implements layout.

For this work, a microprocessor specification is an ISA, and an implementation is its RTL implementation. These levels are most closely related to computer architecture, and are most affected by instruction frequencies. Lower levels are more distant from architectural issues, and involve more general computer aided design problems such as logic synthesis and compaction. Appendix D discusses these lower levels of design.

An ISA is the specification of the external behavior of a microprocessor, and its RTL implementation is its highest circuit oriented, structural implementation.

For example, consider the ISA definitions of the add instruction in architecture manuals for three current microprocessor architectures, the SPARC, the MIPS, and the BAM.

```
r[rd] <- r[rs1] + (r[rs2] or sign extnd(simm13))
        ([SPARC], page 6-7)
GPR[rd] <- GPR[rs] + GPR[rt]
        ([MIPS], page A-9)
rk <- ri + rj
        ([BAM-Manual], page 17)
```

These quite similar definitions are in register transfer form, the form usually used for specifying ISAs. Behavior is defined in terms of user-visible registers (register files in this case, with different indexing notations) and programming-language-style operations on those registers. The plus and assignment operators are abstract -- they are not tied to specific ALUs, adders, or buses.

Now consider an RTL implementation of one these processors. The Cypress CY7C601 SPARC implementation's integer unit block diagram, depicted in Figure 1-1 (from [SPARC], page 2-1), shows among other elements an arithmetic-logical unit, a shifter, and a register file, connected by associated buses. This diagram illustrates the implementation's commitment to a basic set of hardware elements and interconnections.

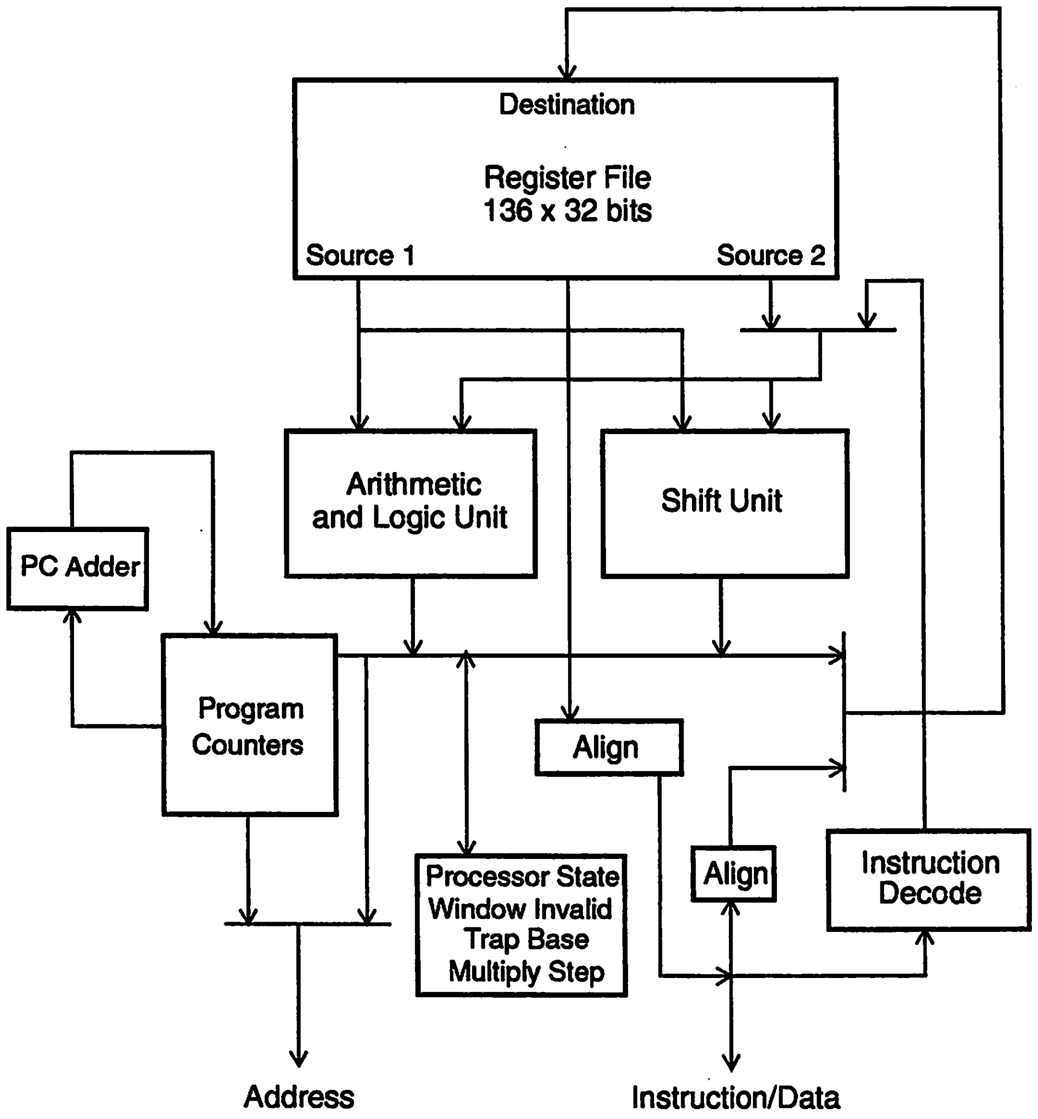


Figure 1-1: A SPARC Data Path

An important part of the implementation that does not appear in the diagram is control. The diagram shows the processor's data path, but the signals that control the data path elements (that cause values to be stored in registers, for example, or that select the function to be performed by the ALU) are missing. The control path, controlling which and when operations are performed, is as important to the implementation as the data path. Control paths are implemented with some type of finite state machine.

1.2. Generating Implementations Automatically

Hayes succinctly describes the problem of generating a processor implementation at the register transfer level from an ISA specification, and identifies it as the register level design problem: "Given a set of algorithms or instructions, design a circuit using a specified set of register level components which implements the desired functions while satisfying certain cost and performance criteria" ([Arch-Hayes], page 141).

This register level design problem is a parameterized optimization problem. In the case of microprocessors, performance criteria are usually expressed in terms of speed or delay constraints, and cost in terms of chip area constraints.

This problem is difficult, and contains NP-complete problems embedded in it. There is no single accepted technique for solving it efficiently and well. As Hayes observes, "Lacking appropriate mathematical tools, register level design methods tend to be heuristic and depend heavily on the designer's experience" ([Arch-Hayes], page 141).

The register level design problem is an instance of the more general high-level hardware synthesis problem, which has been a focus of computer aided design research.

2. What is high-level hardware synthesis and how is it performed?

High-level hardware synthesis addresses the application independent form of the design problem described above for microprocessors. It maps general behavioral specifications to hardware structure ([Tutorial]). It is not limited to the microprocessor domain.

Considerable high-level synthesis work has been done. Several high-level synthesis systems have been constructed. Since the problem is hard, all such systems have various limitations, and have various styles.

This prior work is reviewed in Chapter 2, providing an extensive answer to the above subquestion.

The basic high-level synthesis system created as part of the work described here is presented in Chapters 3, 4, 5, and parts of 8.

3. How can quality be improved with instruction frequencies?

This is the fundamental question posed by this work. Various usage-based common case optimization techniques are explored and evaluated, and are described in Chapters 6, 7, and 8. These chapters more fully answer this subquestion.

In general, this work 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. Mechanisms are needed to, first, compute these weights and orderings, and, second, to use the weights and orderings in synthesis decisions.

4. How can the quality of improvements be measured?

Different microprocessor implementations can be generated from the same specification. Thus, for this work, unoptimized, base line implementations and optimized implementations are synthesized from the same specification. Their quality can then be compared through the use of appropriate area and speed metrics.

The primary metrics conventionally applied to microprocessor implementations, and used in this work, are relative chip area and speed. Speed can be described in a number of ways (including clock rate and instructions executed per second); a performance metric is employed here, essentially cycles per instruction, and is described in Chapter 6.

The experimental technique is fully described in Chapter 9, and results are presented in Chapters 10, 11, and 12.

5. Extended Table of Contents

The experiments constructed to answer the fundamental questions raised above entailed the definition and construction of a number of components:

(1) a hardware specification language, suitable for the definition of microprocessors;

(2) experimental microprocessor specifications;

(3) benchmark applications for instruction frequencies;

(4) a hardware simulator, and other mechanisms, for the collection and management of instruction frequency data;

(5) a translator front end for the hardware specification language, able to translate it into appropriate internal forms;

(6) a translator back end, capable of constructing hardware from the front end's internal forms, and using instruction frequencies in the process; and

(7) a well-characterized functional unit library.

These components and experiments are described in thirteen chapters. Five appendices complete the work, describing some items in more detail, and presenting peripheral issues.

Chapter 2 reviews previous work in high-level hardware synthesis.

Chapter 3 introduces the Viper system, used in the experiments.

Chapter 4 describes the system's hardware specification and simulation language.

Chapter 5 presents the front end translator.

Chapter 6 discusses the collection and management of instruction frequency data, and the general nature of common case optimizations.

Chapter 7 describes common case scheduling transformations. The specific technique used is trace scheduling, applied to the hardware synthesis domain.

Chapter 8 describes common case data path construction. The technique is a new one based on dividing functional units into two categories, those required to implement a design, and those that will optionally improve its performance; optional functional units are allocated in decreasing order of frequency.

Chapter 9 presents an overview of the experiments, focusing on the different synthesis paths through the system.

Chapter 10 describes a simple microprocessor and its synthesis.

Chapter 11 describes the 6502 microprocessor and its synthesis.

Chapter 12 describes the BAM microprocessor and its synthesis.

Chapter 13 describes extensions to Viper to generate pipelined designs.

Chapter 14 presents general conclusions and potential future work.

Appendix A presents variants of the Simple Machine 1 microprocessor.

Appendix B describes the instruction frequency data used in the experiments.

Appendix C describes the process of adding operators and associated functional units to Viper.

Appendix D generally describes the various levels of the Advanced Silicon compiler in Prolog (ASP) system, of which Viper is a part.

Appendix E analyzes the experience of using Prolog for this work.

In addition, a companion document, *A Prolog-Based High-Level Hardware Synthesis System: Source Code and Examples* ([Viper]), contains: the Prolog source code for the Viper system; the Viper functional unit library; a complete Prolog specification of the 6502 microprocessor; two complete Prolog specifications of the BAM microprocessor; and transcripts of the system being used to synthesize microprocessors. It is publicly available in postscript form, via anonymous FTP, from ic.berkeley.edu.

This document and its companion were originally combined and filed as a U.C. Berkeley Ph.D. dissertation.

Chapter Two: High-Level Hardware Synthesis

This chapter describes previous work in high-level hardware synthesis, highlighting limitations and opportunities.

1. Historical Introduction

The process of high-level hardware synthesis, the generation of hardware structure from a behavioral specification (introduced in the previous chapter), has been under investigation for some time. Work at IBM was first reported in 1969 [ALERT], and the first CMU system was described in 1979 [CMU-CAD]. Forty systems are documented in [Survey], which is not exhaustive. Four systems alone have been developed at CMU ([CMU-CAD], [CMU-DA], [SAW-Intro], [Emerald]), and three efforts at AT&T are continuations of that work ([A2S], [Bridge], [CHARM]). Four have been developed at IBM ([ALERT], [YSC], [V], [HIS]). Long, ongoing efforts exist at USC ([ADAM]), the University of Kiel ([MIMOLA]), IMEC (the Cathedral DSP systems -- [Cathedral]), and the University of Illinois ([Chippe], continued at UC Irvine). The MacPitts/ MetaSyn/Silc system has existed for 10 years in various incarnations ([MacPitts], [Silc]), through various attempts to commercialize the technology. Books have been written on the subject ([SiliComp], [HLVS]).

The synthesis process itself is quite complex, and is not fully understood or well characterized. Despite the amount of research that has been done, further research continues to be done (it occupied four sessions at the 1991 Design Automation Conference).

2. High-Level Synthesis Tasks

High-level 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.

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. See Chapter 5 for a list of such optimizations used in current systems (and the ones used in this work).

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, and are discussed below.

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. Intra-Block Scheduling

The basic technique used for most intra-block scheduling is list scheduling (see [Survey] -- it is the most popular high-level synthesis scheduling technique), which was developed for microcode compaction and has been empirically validated for that domain (see [Trace]).

In list scheduling, a data dependency graph of operators is constructed, indicating which operators are dependent on (the results computed by) others (the "list" in list scheduling refers to these lists of dependent operators). The graph is then traversed sequentially, each operator assigned a time step one greater (in the best case) than that of the last operator on which it depends.

Sometimes more operators can be assigned to a time step (based only on data dependencies) than hardware will support (two additions, for example, may be ready, but only one ALU may be available). In that case, the scheduler must give priority to some operators, and delay the others.

Two priority functions are commonly used. The first uses the operator dependency lists ([A2S], for example). If delayed, an operator can potentially delay all its dependent operators in turn. Thus this priority function gives priority to the operators with the greatest number of dependents.

The second employs the concept of mobility ([Chippe-Micro]), or, equivalently, freedom ([MAHA]). Mobility refers to the number of potential time steps to which an operator can be assigned, between when the operator's data dependencies are met (when its operands are ready) and when the operator's result is required by other operators (specifically, the difference between an operator's as-late-as-possible and as-soon-as-possible time steps). If delayed, a less mobile operator has fewer alternate time steps. Thus this priority function gives priority to the operators with the smallest mobility.

List scheduling with mobility is used in this work in an intra-block scheduling phase (see Chapter 8, step 6).

A variant of mobility-based scheduling, called force-directed scheduling, was developed specifically for high-level synthesis ([HAL-FDS]). It differs from list scheduling in one important respect. List scheduling scans operators in order of dependency (using the dependency lists and processing them sequentially). Force-directed scheduling, in contrast, scans operators in order of mobility, least mobile first, thus giving mobility absolute priority. The name of the technique comes from the analogy of mobility to a compressed spring -- the smaller the mobility the greater the force on the spring. Processing operators in order of least mobility thus causes each step to result in the greatest reduction of force. Force-directed scheduling is now used by 3 systems (see [Survey]).

An interesting list scheduling alternative can be found in the HYPER system ([HYPER]), which performs scheduling after allocation. The above techniques use priority functions involving computations based on specification operators; HYPER uses a priority function based on allocated functional units. At each time step the system computes, for each type of hardware resource, the ratio of available resources over required resources, and schedules the resource with the smallest ratio (which is therefore in the shortest supply).

4.2. Inter-Block Scheduling

Transforming a design to increase concurrency by moving operations between basic blocks is a complex process, and potentially expensive in duplicated operations. It has been studied to some extent, however, because of the possibility of enhanced performance.

4.2.1. Flamel

The most important system, for this dissertation, is the Flamel system ([Flamel], named after a medieval alchemist), which compiles a restricted subset of Pascal to hardware. Significantly, it combines basic blocks using a set of transformations, and decides whether or not to apply a given transformation using execution counts derived from benchmarks.

Specifically, Flamel uses five transforms.

- Linemerge merges two basic blocks, one following the other, into one.
- Altmerge merges the three blocks of an if-then-else (the test, the true arm, and the false arm) into one. It does this by attaching multiplexers, driven by the output of the test, to the outputs of the true and false arms.
- Unroll unrolls a loop one iteration. It connects the loop outputs to the loop inputs and uses a multiplexer driven by the loop index to select which input is used.
- Fullunroll is used to unroll a loop when the number of iterations is known. It operates like unroll, but uses as many loop copies as there are iterations.
- Tat-to-tab converts a test-at-top loop to a test-at-bottom loop, which is the form that unroll and fullunroll require.

Note that transforms can be applied to the results of other transforms.

In most cases the nesting structure of blocks determines the order in which transforms are applied. There are cases, however, where the order is ambiguous; Flamel imposes fixed priorities in those cases (always choosing an altmerge over a linemerge, for example). The result of these fixed priorities is significant -- Flamel never has to decide *which* transform to apply, it only decides *whether* to apply one. The result is a tree of transforms and blocks, with the initial blocks as leaves, and the results of transforms as internal nodes (see Figure 4 in [Flamel]).

The question that Flamel must then address for each transform in the tree is, is it cost effective to apply that transform? To answer the question Flamel computes the hardware cost and weighted performance (weighted by execution frequency) of every block in the transform tree, including initial blocks and transformed blocks. It then does a preorder traversal of the transform tree. If a transformed block is faster, it uses it; otherwise it does not apply the transform, but instead moves on to the transform's children in the transform tree. This method results in globally optimal execution times.

Flamel is a model for inter-block optimization: it uses powerful transformations, and guides their application with execution frequencies.

Flamel does, however, have limitations.

- It is limited to five optimizing transforms, which do not handle all control structures.
- In order to generate an unambiguous transform tree, necessary for evaluating the utility of transforms, it is committed to a predefined priority of transforms.
- It handles the movement of operators between blocks (effectively the combining of blocks) by introducing multiplexers in the data path, along with associated value selection logic. This is more complex and costly than adding states to the controlling finite state machine, and therefore can produce designs that are far from being globally optimal.

4.2.2. The SUGAR system (early version)

Researchers at CMU have extensively investigated inter-block transformations ([CMU-BLT], [CORAL], [CORALII], [CMU-DRT], [SAW]). The focus of this work has been interactive, user-driven transformations. Only one effort automated the process.

An early version of the SUGAR system ([SUGAR]) used ordering information to optimize case statements. It allowed the user to annotate manually a specification with ordering information, attaching priorities to the arms of case statements. Execution paths through the case arms that did not meet a general delay constraint were optimized in the order specified by the user, by moving operations out of the case arms in a manner similar to trace scheduling.

This approach is suggestive but limited.

4.2.3. The VSS System

The VHDL Synthesis System (VSS) ([VSS]) performs inter-block scheduling using loop unrolling and percolation scheduling ([VSS-PS]), applying these programming language techniques to the domain of high-level synthesis.

Percolation scheduling was developed to increase concurrency in programs executed on parallel hardware. It consists of a set of (four) atomic transformations that move operations over block boundaries, while preserving correctness. These atomic transformations are applied under the guidance of various optimizing heuristics. The heuristics are typically driven by larger structural program features (such as conditionals and loops).

The VSS heuristics are concerned with loop optimization, and use the technique of loop unrolling. The key to their operation is the observation that, in most cases, loops can only be unrolled a few times before data dependencies make further unrolling unproductive. The VSS heuristics unroll a loop until a pattern of operations and dependencies emerge that is stable under more unrolling. The result is provably optimal loop execution.

Conditionals in the loop are handled, but substantial constraints limit the guarantee of optimality (involving the distance operators can be moved, and regularity in the application of transforms).

The VSS technique is primarily aimed at DSP applications (see the examples given), with computation-intensive loops. It is not general, and is further limited in its ability to process conditionals.

4.3. The Scheduling Opportunity

All the above inter-block scheduling techniques have limitations, as noted above. The primary one common to all is restricted transformation ability. The transforms are pattern-specific and limited in their application. In addition, the VSS optimizations are not driven by usage information.

Another system not so limited is the HIS system ([HIS]), which performs general inter-block scheduling. It optimally schedules each execution path through the design, and then optimally combines the individual path schedules. It is, however, computationally quite expensive. Each scheduling step is formulated as a clique partitioning problem (see scheduling below), which is NP-complete. A solution thus requires the solving of n+1 NP-complete problems, where n is the number of paths through the design.

It would be worthwhile to investigate a general, efficient, inter-block usage driven scheduler. In particular, high-level synthesis is similar to very long instruction word (VLIW) compilation (see [Bulldog] and [Arch-H&P]), with the instruction word width (the available resources) not fixed in advance. It would be appropriate to apply the inter-block trace scheduling technique developed for VLIW compilation to high-level synthesis. See Chapter 7.

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.

A wide variety of specific allocation methods have been used. These methods tend to be somewhat idiosyncratic, depending on paradigm (such as expert systems, or graph algorithms), target hardware (available functional units), problem domain (such as microprocessors or digital signal processing), relation to scheduling, method of register allocation, and method of bus allocation. In particular, register and bus allocation, involving storage and connectivity, present different problems from functional unit allocation and binding; nonetheless, all three types of allocation are related, and interact.

In addition, most systems perform local optimizations and transformations, recognizing special case operators (such as incrementers). These optimizations tend to be idiosyncratic (based on the available functional units). An effective synthesis system should employ some of these optimizations. See Chapters 5 and 8 for a description of the ones used in this work.

Note that allocation is related to scheduling (see [Tutorial]). Functional unit sharing is predicated on the knowledge that two operators bound to the same unit will not use it at the same time -- that they are not scheduled in the same cycle. Most allocation techniques assume at least a preliminary schedule. Some techniques ([MAHA], for example) modify the schedule.

Also note that, as mentioned above, allocation in general refers to both the selection of hardware elements and the mapping of operations to those elements. The second process is sometimes referred to as binding, to distinguish it from the first.

5.1. Specific Allocation Techniques

The allocation problem can be formulated as the process of trying to map operators assigned to different time steps to the minimum number of functional units. It is then very similar to the register allocation problem for compilers, which tries to map variables with different lifetimes onto the minimum number of registers. Graph coloring and clique partitioning are different graph-based solutions to the same problem, and are used in high-level synthesis (see, for example, [YSC-Design] and [HIS-DP]).

The classic system to use this technique is FACET ([Facet], [Emerald]), which uses clique partitioning. Recall that a clique is a complete subgraph (with every node connected to every other node). The formulation is simple: construct a graph where the nodes are operators and the edges indicate possible sharing of hardware (operators assigned to different time steps that can use the same functional unit). The minimum hardware allocation is defined by the minimum number of cliques, where each clique represents a set of operators mapped to a functional unit. The clique partitioning problem, which determines that minimum number, is NP-complete (problem GT15 in [NP]). There are, however, good approximate algorithms (see [Facet]). The same technique can be used for other cases of maximizing potential sharing, variables and registers, and data transfers and buses in particular. The technique (and graph coloring, formulated similarly, GT4 in [NP]) is used in several systems (see [Survey]).

Another system that maps sets of operations onto functional units is the CHARM system at AT&T [CHARM]. Unlike FACET, which attempts a global, simultaneous optimization, CHARM merges sets of operations pairwise, choosing the most cost-effective merge, until no more merges are possible. The sets of operations merged are disjoint -- each operation is executed in a different time step. The system iteratively tries merging all pairs of sets of disjoint operations, choosing the merge with the greatest hardware savings (cost improvement) at each step. The cost improvement is the difference in cost between the two sets implemented with separate functional units and with the same functional unit, taking into account interconnection costs. The Flamel system is similar ([Flamel]), but uses a different cost metric.

Some systems allocate individual operations sequentially, rather than in sets. The EMUCS allocator ([EMUCS], part of SAW [SAW]) works in this manner, iteratively constructing a data path operation by operation. At each iteration step EMUCS constructs a table containing all the costs (for all possible implementations) of implementing each of the as yet unimplemented operations. The system then implements the operation with the greatest difference between cheapest and second cheapest cost (which can be thought of as the cost of not implementing the operation this iteration).

Another system that allocates operations sequentially is the MAHA allocator from USC [MAHA]. Rather than process operations on the basis of implementation cost, as EMUCS does, it processes them in the order they were scheduled. Operations are processed block by block, cycle by cycle. At any cycle, the operation with the least freedom (see intra-block scheduling, above) is allocated first. If an operation cannot be implemented, because doing so would exceed hardware cost constraints, the operation is delayed a cycle, with the expectation that the necessary hardware may be free in that next cycle.

In contrast to the above local, greedy approaches, some systems use integer and linear programming to solve the scheduling and allocation problems. Unfortunately, these solutions are quite cumbersome to define, and are inefficient to solve ([LPS], [MIMOLA]).

Two other systems, SUGAR (described in [SAW]), and the Design Automation Assistant ([DAA]), follow the opposite strategy, being very specific instead of very general. Rather than trying to solve general allocation and scheduling problems, they fundamentally use pattern matching. In the case of SUGAR, the patterns are compiler-based (code templates, for example); with DAA they are rules in an expert system (implemented using OPS5). They both are tuned to generate processors and processor data paths, and are optimized for certain technologies (2-bus VLSI data paths for SUGAR, TTL for DAA). They are thus rather limited.

Some systems add an optimization stage after translation and before scheduling and allocation. This stage is in broad concept similar to the operation merging techniques above, but it is employed before allocation, and merges behavioral operators into clusters that will likely result in cost effective allocation, using various prediction and partitioning metrics ([CMU-Cluster], [BUD], [A2S], [APARTY], [YSC-Partition], [DFBS]). The process is analogized to floorplanning ([CMU-Cluster]).

5.2. The Allocation Opportunity

The above techniques attempt to minimize data path cost. This is not necessarily desirable when designing for performance (that is, overall execution speed). Commonly executed operations are more important than infrequent ones. It would be worthwhile to investigate a greedy, iterative allocation technique that gave priority to common operations. The MAHA operation-by-operation technique would be a reasonable starting point. See Chapter 8.

Chapter Three: An Introduction to Viper

This chapter reviews the ASP hardware synthesis project, and introduces its high-level synthesis component, Viper.

1. The ASP Context

The Advanced Silicon compiler in Prolog (ASP) is a full-range hardware synthesis system that produces layout from behavioral hardware specifications ([ASP-Intro]). It was begun as part of the Aquarius project ([Aquarius]), to provide CAD support and to test Prolog as an implementation vehicle.

The goal of the Aquarius project was to produce high-performance Prolog engines, realized in part with specialized high-quality microprocessors (see [BAM]). Thus the focus of ASP is single-chip microprocessor synthesis, within a design domain of single synchronous clock chips with a single data path and control path.

ASP is automatic, top-down, and integrated. It does not employ interactive user guidance or automated redesign. It is written entirely in Prolog, hence its implementation is both algorithmic and rule-based. In general the system is algorithmic, with rule-based local optimizations.

The ASP system includes (see [ASP-Prototype] and Appendix D): Viper (see below); Topolog, a module generator, which expands data paths to full gate-level netlists and generates gate matrix style layout; Sticks-Pack, a technology independent compactor; a PLA generator (which also performs state assignment); a left-edge-first channel router; and a global placer and router. The general ASP system is described in Appendix D.

2. The Viper System

Viper[1] is the high-level synthesis component of ASP. The paradigms and implementation of Viper have been greatly determined by those of ASP.

Viper 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; the resulting experience with Prolog is documented in Appendix E. As a result of its microprocessor orientation, it is less concerned with optimizing complex expressions (than, for example, DSP oriented systems such as [Cathedral]). The source code for Viper is provided in [Viper].

Viper operates on input specifications written in executable Prolog. The level of specification is designed to be the lowest level that can still be executed directly by a Prolog interpreter. Such constructs as architected registers, bit fields, and a memory interface can be specified (and both simulated and synthesized). Since the general ASP goal is to synthesize microprocessors rather than compile general Prolog to hardware, such language features as recursion and full unification are not supported. The output of Viper is a conventional collection of connected data path elements and a controlling finite state machine.

In general, high-level synthesis in Viper follows the form described in the previous chapter (see [Tutorial]). The primary paths of design flow through Viper are pictured in Figure 3-1.

Along the basic path, Prolog specifications are translated into register transfer level representations (also in Prolog), which are used for interrelated hardware scheduling and allocation, which in turn produce control and data paths (also in Prolog).

Additional paths involve instruction frequency data collection and its subsequent use during allocation and trace scheduling, which itself involves an extra path in which RTL is modified.

1. Viper is not an acronym: it is the first part of the phrase "viperformance hardware synthesizer".

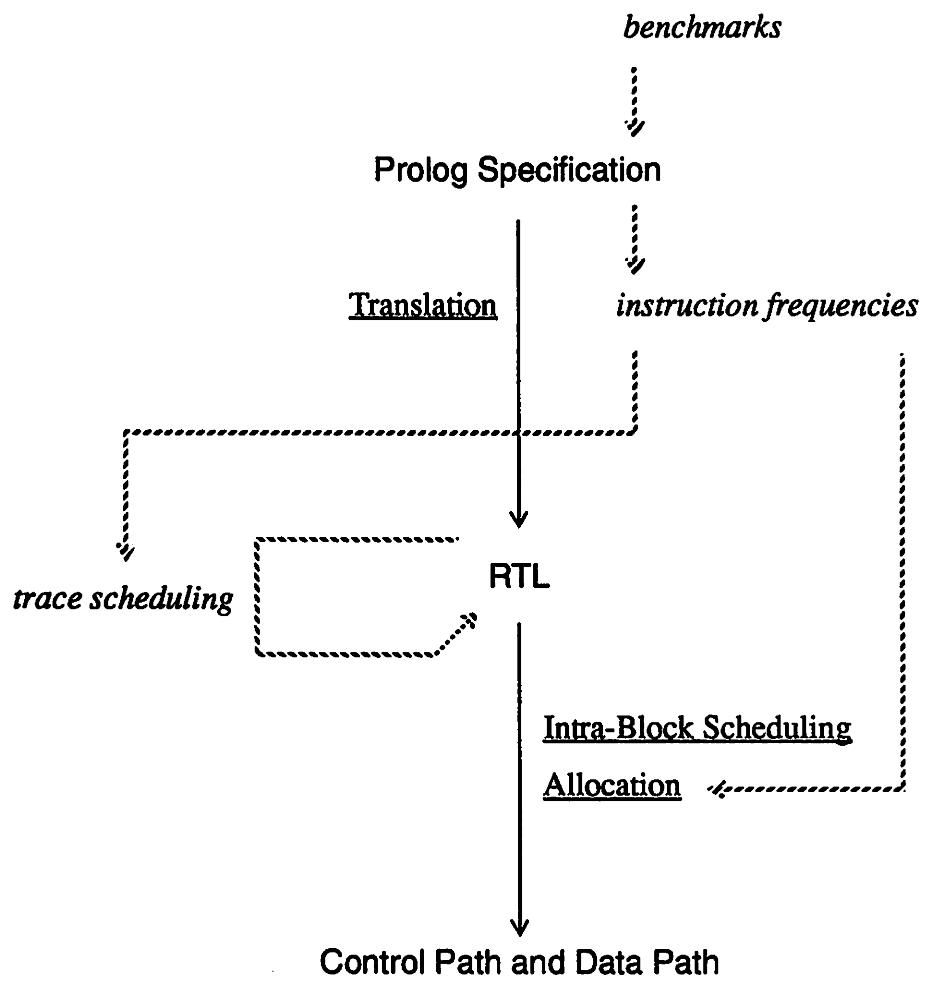


Figure 3-1: Viper Overview

The heart of this work traces these additional paths, and consists of scheduling and allocation techniques, either new or newly applied to high-level synthesis, that are driven by instruction frequencies. These are discussed, along with other necessary mechanisms, in Chapters 6, 7, and 8. The ensuing experiments compare the results of synthesizing microprocessors with and without those techniques. The details of the experimental approach are presented in Chapter 9, and the results in Chapters 10, 11, and 12.

In detail, Viper operates in nine major phases.

(1) It translates the Prolog input specification (Chapter 4) into a data flow graph and a control flow graph (Chapter 5).

(2) It converts the data flow graph into an equivalent set of register transfers (Chapter 5).

(3) It optionally computes instruction frequencies (Chapter 6).

(4) It optionally trace schedules, using instruction frequencies (Chapter 7).

(5) It computes preliminary as-soon-as-possible dependency-based intra-block schedules (Chapter 8).

(6) It performs a preliminary allocation of functional units, guided by instruction frequencies (Chapter 8).

(7) It performs a final allocation and scheduling, again guided by instruction frequencies (Chapter 8).

(8) It connects the allocated functional units with buses (Chapter 8).

(9) It packages the information generated in the prior phases into a form suitable for use by lower level tools (Chapter 8).

Chapter Four: Behavioral Specification and Simulation in Prolog

The goal of the hardware language design undertaken in this dissertation was to develop a straightforward specification medium for a variety of existing microprocessors, within the larger context of the ASP project (see Chapter 3), of which this work was part. Thus the level of specification is relatively low (roughly equivalent to ISPS [ISPS]), and the base language is Prolog ([Prolog]).

The details of this specification medium follow, preceded by a general description of the specification languages that have been used in high-level synthesis. This material is presented so that examples and microprocessor specifications in subsequent chapters can be understood. Further, the form of behavioral description affects the optimizations and translation problems that a synthesis system must address.

1. Specification for Synthesis

Specification languages used by high-level synthesis systems are of two root types, and those roots have been modified in two basic ways.

The first language type is based on some familiar programming language. Such foundations as LISP ([MacPitts] and [Silc]), OCCAM ([OCCAM]), Pascal ([Flamel], [Chippe], and [MIMOLA]), and C ([HERCULES] and [HAGGLER]) have been used. Typically, programming languages are chosen because they are familiar, are rooted in simple, general paradigms (such as LISP and OCCAM), and are easily modified for hardware specification.

The second type of language is based on a hardware simulation or specification language. VHDL ([VSS-VHDL]) and ISPS ([SAW]) are examples. Simulation languages are simple to use in some ways because they have been designed to describe hardware, but cause problems because they usually allow the specification of hardware structure as well as behavior, thus simultaneously supporting multiple levels of abstraction. This is difficult for synthesis systems to handle, which, in general, for simplicity, map a single, higher level of abstraction into a lower one.

The two categories of language modifications are restrictions and additions. Restrictions are introduced to make hardware more easily or efficiently synthesized, and usually involve restricting data types, with some additional limits on control constructs and operators (see, for example, [MacPitts] and [Flamel]). Additions are made for the specification of hardware-oriented constructs, such as interface signals, concurrency, and resource constraints (see, for example, [MacPitts] and [Chippe]).

A noteworthy case of language restriction is the VSS system ([VSS-VHDL]), which uses the VHDL hardware simulation language. This system faces serious language subset problems because VHDL supports mult-level simulation, and allows specification at more than the purely behavioral level. The VSS system implementors decided not to restrict specifications to the behavioral level; instead, they allow restricted specification at all levels, and use the system to combine multi-level specifications.

2. Using 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 Viper 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. This general approach is similar to that taken with the MacPitts system for LISP and with the Flamel system for Pascal.

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. Higher level optimizing transformations ([SAW]) can be layered on top of this basic functionality.

3. Restrictions

The microprocessor specification domain of ASP (the highest domain) makes standard Prolog ([Prolog]) a reasonable choice as a specification language. Multiple asynchronous finite state machines, explicit parallelism, and detailed off-chip interface descriptions need not be supported. The specification domain is also constrained by ASP'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:

(i) the specification must be deterministic (with only shallow backtracking),

(ii) it can contain no lists or structures,

(iii) it must be only tail-recursive, without true recursion, and

(iv) 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.

4. 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 four classes:

(i) support for maintaining global state (in the form of registers and register files),

(ii) additional hardware-oriented operators (such as add with carry),

(iii) support for system functions (such as interfaces and memory), and

(iv) support for simulation.

The next section presents basic extensions in conjunction with a simple example; subsequent sections enumerate the extensions in more detail.

5. Basic Extensions and Specifications

This section describes registers, fields, and memory, and their simulation, through a simple example processor specification.

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.

In addition, a built-in is provided to change a specification's state. Individual registers are modeled as master-slave components, so that accesses to a register retrieve its old value, distinct from its new value, which is specified by set. At some point the new value must replace the old one; this transition, or clocking, occurs simultaneously for all registers at intervals specified by the user with the *stateUpdate* built-in. Thus in the fragment

```
set(pc, 0),
set(ac, 1),
stateUpdate,
set(ac, -1),
access(ac, A1),
stateUpdate,
access(pc, P),
access(ac, A2), ...
```

A1 will be 1 and A2 will be -1. Note that old values, unchanged by set, are carried over, so that P will be 0. This general concept of state is similar to that developed by researchers interested in adding objects to Prolog ([Objects-Intensions], [Objects-Logical]).

Also 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.

A final additional set of built-ins provides a simple memory interface, an important (microprocessor oriented) connection to the outside world. 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.

Now consider an example (this example is a recasting of one presented in [ASP], which involved an earlier, more limited model of variables and registers). It is the definition of a simple Von Neumann machine, defining the operation of microprocessor instructions. Individual instruction specific clauses are contained in a recursive instruction executing definition.

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

An add instruction is defined with an execute clause.

```
execute(add) :- !,
    access(memDR, address, X), set(memAR, X),
    mem_read,
    access(memDR, T), access(ac, AC), A is T+AC, set(ac, A).
```

This specification itself is relatively abstract, compared to many hardware specification and simulation languages. Explicit concurrency, timing, and hardware entities (such as buses) are not present.

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

In fact, simulation can take two forms, depending on exactly how register fields are modeled. Fields can be modeled either as Prolog structures, or as bits in a single integer word. For example, from the above specification the memDR could either be a structure with opcode and address fields, so that an add instruction would be represented as

```
inst(add, 8).
```

(adding location 8 to ac), or it could be an integer, so that the same add instruction would be simply be the bit pattern

```
1032
```

assuming an add opcode was represented as a 1 in an opcode field starting 11 bits from the rightmost bit. The former higher-level form is appropriate for processors for which an assembler does not exist, while the latter is suitable for lower-level simulation of bit images.

For comparison, the exact field declaration forms and the symbolic and numeric register declarations for the microprocessor example above are presented below.

```
% symbolic
stateField(<register-name>, <structure-name>,
        <field-name>, <field-index>).
% numeric
stateField(<register-name>, <field-name>, (<first-bit> - <last-bit>).
% symbolic
stateRegister(ac, 16).
stateRegister(pc, 16).
stateRegister(memAR, 16).
stateRegister(memDR, 16).
stateField(memDR, inst, opcode, 1).
stateField(memDR, inst, address, 2).
% numeric
stateRegister(ac, 16).
stateRegister(pc, 16).
stateRegister(memAR, 16).
stateRegister(memDR, 16).
stateField(memDR, opcode, (15-10)).
stateField(memDR, address, (9-0)).
```

With these definitions, we can define and execute a program with the above example processor specification. The facts

```
% symbolic
mem(0, inst(add, 8)).
mem(1, inst(add, 9)).
mem(2, inst(stor, 10)).
mem(3, inst(halt, _)).
```

```
    mem(8, 2).
    mem(9, 3).
    % numeric
    mem(0, 1032).% 1(1024)+8 inst(add, 8).
    mem(1, 1033).% 1(1024)+9 inst(add, 9).
    mem(2, 5130).% 5(4096+1024)+10 inst(stor, 10).
    mem(3, 0000).% 0 inst(halt, _).
    mem(8, 2).
    mem(9, 3).
```

define a program and its data; Starting at location 0, add two numbers, 2 and 3, in locations 8 and 9, and store the result in location 10.

For completeness, the memory system simulator is

```
    mem_read :-
        access(memAR, Loc),
        mem(Loc, Data),
        set(memDR, Data).
    mem_write :-
        access(memAR, Loc),
        mem(Loc, _), !,
        retract( mem(Loc, _) ),
        access(memDR, Data),
        assert( mem(Loc, Data) ).
    mem_write :-
        access(memAR, Loc),
        access(memDR, Data),
        assert( mem(Loc, Data) ).
```

6. 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.

There are additional built-ins that support simulation:

- *stateUpdate* (for changing a specification's state in a master-slave fashion -- see above);
- *setDefault(<register>, <value>)*, which sets a default value for the given register that will be used by the next stateUpdate unless explicitly overridden by a set;
- *preset(<register>, <value>)*, which tests to see if the given register has been set since the last stateUpdate, and if it has returns the value; and
- *statePrint*, which prints out the register state of the simulation.

7. Extensions: Register Files

Many modern microprocessors use register files in addition to, or instead of, individual architected registers ([MIPS], [SPARC], and [BAM]). Viper provides built-ins for creating and referencing these 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. Such references can be modeled in two ways, based on timing considerations, both supported by Viper. The two models correspond to different possible register file hardware implementations. In both models, register files have separate, and explicit (and potentially multiple), read and write ports, each of which has an associated index.

The first model has the register index presented to the register file in one cycle, and the value read from or written into the file in the *next* cycle. The second model has the index presented and the value read or written in the *same* cycle.

The first, separate cycle model requires no new built-ins. An ordinary set built-in can be used to set up the index, and an access or set built-in can be used to reference the register value. The model does, however, require two new set and access operand forms, for the index and the value. These forms are, respectively,

port(<file-name>, <port-name>), ...
% index reference
reg(<file-name>, <port-name>), ...
% value reference

The port name refers to the read or write port.

Thus register 10 in register file r is read via

```
set(port(r, read), 10),
access(reg(r, read), ReadValue), ...
```

and is written with

```
set(port(r, write), 10),
set(reg(r, write), WriteValue), ...
```

The second, same cycle model requires two new built-ins. 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

In this model, 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), ...
```

8. 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. Such analysis can easily be layered on top of Viper. The operators that have been added are:

- *addc(<value-1>, <value-2>, <carry>, <result>)* (add with carry),
- *subc(<value-1>, <value-2>, <carry>, <result>)* (subtract with carry),

- *xor(<value-1>, <value-2>, <result>),*
- *ash(<value-1>, <value-2>, <result>)* (arithmetic shift),
- *rotate(<value>, <carry>, <type>, <result>)* (this is a one bit rotate, with *<type>* either << or >>),
- *min(<value-1>, <value-2>, <result>),* and
- *max(<value-1>, <value-2>, <result>).*

Another operator has been added as an optimization. It assigns the boolean output of a comparison, or similar operation that produces a boolean value. For example, the fragment

```
t(Data) :- Data = 0, !, set(psw, zeroflag, 1).
t(_) :- set(psw, zeroflag, 0).
```

sets the zeroflag of the psw if Data is 0. When this is translated it creates three basic blocks (one for the test, one for the true arm, and one for the false arm). This can be collapsed to one basic block with the test value (*tval*) operator,

```
t(Data) :- tval(isZero, Data, Bool), set(psw, zeroflag, Bool).
```

which generates the boolean value Bool, which can then be used a regular value. This optimization reduced the number of blocks in the 6502 specification from 213 to 93. As with the other operators, this tval form could be recognized via analysis, but Viper takes as input the lowest level of executable specification. The tval operator has the general form

tval(<test>, <test-value>, <result-boolean>)

where *<test>* can be isZero, isNeg, isOver, or isCarry.

9. 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.

Registers can also be used for input and output, as latches. Such registers are declared very much like normal registers. A third argument is used in the *stateRegister* declaration to indicate direction (in or out).

Finally, priority encoders can be defined, so that sequential testing of flags can be avoided. Consider, for example, the interrupt tests

```
interrupt :-
    test(halt, 1), !,
    ...
interrupt :-
    test(reset, 1), !,
    ...
interrupt :-
    test(int, 1), !,
    ...
```

where halt, reset, and int are stateInterface lines. These tests will be performed sequentially. They should be performed in parallel, and in priority order (halt being most important). To support this, for both simulation and synthesis, the *statePriority* built-in is provided, which in the above case would be expressed as

```
interrupt :-
    statePriority([halt, reset, int], Priority),
    interrupt(Priority).
interrupt(halt) :-
    ...
interrupt(reset) :-
    ...
interrupt(int) :-
    ...
```

During simulation, the Priority variable is set to the symbol corresponding to the line that is raised. That variable is then matched in the following unification to the proper interrupt clause. When synthesized, the statePriority construct enables the appropriate interrupt line.

Note that, as with other hardware motivated features, this one could be recognized via analysis, but this is the lowest level of executable specification.

10. Extensions: Simulation

Additional built-ins have been defined to support simulation. Several have been discussed already: stateUpdate, setDefault, preset, and statePrint. Two additional ones are

- *noSyn*, which, when used as the first goal of a clause, blocks that clause from being synthesized, and
- *stateCount(<count>)*, which returns the number of cycles, or stateUpdate goals, executed so far.

The *noSyn* built-in is particularly useful for clauses that perform error checks during simulation.

In addition, of course, simulation requires executable versions of the register manipulation procedures, the operators, and the system support functions.

11. A BNF Definition

In summary, the BNF definition of the syntax of Viper's specification language is:

```
<specification> ::= <declarations> <procedures>

<declarations> ::=
        <declaration>
        | <declarations> <declaration>

<declaration> ::=
        stateRegister(<name>, <width>).
        | stateRegisterSet(<name>, <width>, <size>).
        | stateField(<register>, <name>, <range>).
        | stateInterface(<name>, <constant>).

<procedures> ::=
        <procedure>
        | <procedures> <procedure>

<procedure> ::= <simple-procedure> | <tagged-procedure>

<simple-procedure> ::=
        <simple-clause>
        | <simple-procedure> <simple-clause>
<simple-clause> ::= <simple-head> :- <clause-body>.
<simple-head> ::= <name> | <name>(<argument-list>)

<tagged-procedure> ::=
        <tagged-clause>
        | <tagged-procedure> <tagged-clause>
<tagged-clause> ::= <tagged-head> :- <clause-body>.
<tagged-head> ::= <name>(<tag>) | <name>(<tag>, <argument-list>)
```

```
<argument-list> ::=
        <variable>
        / <argument-list>, <variable>
<clause-body> ::= <basic-clause-body> / noSyn, <basic-clause-body>
<basic-clause-body> ::=
        <goal>
        / <basic-clause-body>, <goal>
<goal> ::=
        <name>(<argument-list>)
        / <conditional>
        / <variable> is <expression>
        / <viper-goal>
        / !
        / fail
<conditional> ::=
        <expression> =:= <expression>
        / <expression> =\= <expression>
        / <expression> > <expression>
        / <expression> < <expression>
        / <expression> =< <expression>
        / <expression> >- <expression>
<expression> ::=
        <constant>
        / <variable>
        / - <expression>
        / <expression> + <expression>
        / <expression> - <expression>
        / <expression> /\ <expression>
        / <expression> \/ <expression>
        / <expression> << <expression>
        / <expression> >> <expression>
        / <expression> \ <expression>
        / xor(<expression>, <expression>)
        / ash(<expression>, <expression>)
        / min(<expression>, <expression>)
        / max(<expression>, <expression>)
<viper-goal> ::=
        mem_read
        / mem_write
        / mem_read(<port>)
        / mem_write(<port>)
        / access(<register>, <variable>)
        / access(<register>, <field>, <variable>)
        / set(<register>, <variable>)
        / set(<register>, <field>, <variable>)
        / test(<register>, <constant>)
        / test(<register>, <field>, <constant>)
        / tval(<test>, <variable>, <register>)
        / addc(<variable>, <variable>, <carry>, <result>)
        / subc(<variable>, <variable>, <carry>, <result>)
        / rotate(<variable>, <variable>, <rotate>, <result>)
        / rval(port(<name>, <tag>), <index>, <variable>)
        / wval(port(<name>, <tag>), <index>, <variable>)
        | stateDefine
        | stateInitialize
        | stateList
        | statePrint
        | stateCount(<variable>)
        | stateUpdate
        | statePriority([<tag-list>], <variable>)
<test> ::= isNeg | isOver | isCarry | isZero

<rotate> ::= >> | <<

<tag-list> ::= <tag> | <tag-list> , <tag>

<name> ::= <tag>
<port> ::= <constant>
<field> ::= <tag>
<register> ::= <tag>
        | port(<name>, <tag>)
        | reg(<name>, <tag>)
<width> ::= <constant>
<size> ::= <constant>
<range> ::= <constant> | <constant>-<constant>
<carry> ::= <variable>
<result> ::= <variable>
```

<tag> is a symbol
<variable> is a Prolog variable
<constant> is an integer constant

Chapter Five: Translating Prolog into Register Transfers

This chapter describes the component of Viper that translates Prolog specifications into register transfers. This translator is important because its RTL output is used by Viper's optimizing scheduling and allocation components (and is used for obtaining instruction frequency statistics through simulation). It also performs optimizations that affect the quality of the final design.

It operates in four stages.

- It scans the Prolog input specification for correctness in terms of the subset of Prolog it supports.
- It translates the Prolog into a data flow graph and an associated control flow graph.
- It generates a set of register transfers that embody the data flow graph.
- It maps certain register transfers into simpler ones.

1. Register Transfers

Viper is like other high-level synthesis systems in that it translates input specifications into data and control flow graphs ([Survey]). The microprocessor orientation of ASP, however, emphasizing architected registers and relatively simple expressions, has led to a subsequent change in representation. The data and control flow graphs are then converted into register transfers, which serve as the unit of scheduling. They explicitly represent latch-operation-latch sequential logic.

The data and control flow graphs are needed for translation and analysis. The RTL transfers are used for simulation, scheduling, and allocation. The transfers introduce additional constraints: each data flow operation becomes an atomic transfer involving up to two source registers and one destination register. This tuple (in compiler terms) is easier to manipulate (schedule and pattern match for allocation) than the general graph, and maps readily into the register to register microarchitecture style of microprocessors.

Specifically, a register transfer has the form

```
rtran(<transfer-index>, <basic-block-identifier>),
    <source-register1>,
    <source-register2>,
    <operator>,
    <destination-register>).
```

where *rtran* is a Prolog structure. Either source may be a constant; the second source may be *none*; the operator may be *move* or a memory operation. The transfer index is a unique identifier (making a collection of transfers a relation in the data base). This representation, more structured than its data flow analog, maps readily into the register to register microarchitecture style of microprocessors.

Additionally, specifications are divided into basic blocks (in the compiler sense); each block consists of a consecutive collection of register transfers, and within a block there is no transfer of control. In turn, these basic blocks serve as another, larger unit of scheduling: within a basic block, register transfers can be scheduled solely on the basis of data dependency.

The memory system presented in Chapter 4, for example, fits easily within this context. Consider again the add instruction.

```
execute(add) :- !,
    access(memDR, address, X), set(memAR, X),
    mem_read,
    access(memDR, T), access(ac, AC), A is T+AC, set(ac, A).
```

Its register transfer representation appears as

```
rtran(5, block(4), field(memDR, address), none, move, memAR).
rtran(6, block(4), memAR, none, mem_read, memDR).
rtran(7, block(4), memDR, ac, +, ac).
```

(assuming it occurs in block 4, starting at transfer index 5). The memory read uses memAR as a source and memDR as a destination.

In general, the problem with this register transfer representation is how to represent expressions that use temporaries (that generate values not stored in architected registers). Viper deals with this by explicitly creating temporaries. For example, the fragment

```
... access(regA, A), access(regB, B),
T is (A + B) >> 1,
set(regC, T), ...
```

which adds A and B, shifts the sum right 1 bit, and stores the result in C, would be realized as

```
rtran(10, block(6), regA, regB, +, temp(1)).
rtran(11, block(6), temp(1), constant(1), >>, regC).
```

(assuming it occurred in block 6, starting at transfer index 10). Whether *temp(1)* is implemented as a latch or a simple connection between the adder and the shifter is a matter of delays and timing.

Currently, such temporaries are always implemented as a connection. Such constructs do not occur often in the microprocessor specifications used in this work. For example, in one Prolog version of the 6502, there are 22 such expressions (out of a total of 340 register transfers).

More importantly (since such combinational logic may be on the critical path in the data path), it is assumed that there is enough time in one cycle to read registers, perform computation, and store the result.

The translator also generates control dependency information important to pipelining (see Chapter 13).

2. Compiler Optimizations

Several systems perform various behavioral level optimizations that are based on programming language compiler technology ([CMU-BLT], [CMU-DRT], [SUGAR], [CADDY], [Flamel], and [HERCULES]). Identified optimizations are: inline expansion (BLT, DRT, SUGAR, and HERCULES), procedure formation (DRT), dead code elimination (BLT, DRT, SUGAR, and HERCULES), common subexpression elimination (BLT and SUGAR), constant folding (BLT, SUGAR, and CADDY), constant conditional elimination (HERCULES), block flattening (Flamel and HERCULES), moving loop invariants out of loops (CADDY), various case optimizations (moving code into and out of case arms, case merging, and case flattening) (BLT, DRT, and SUGAR), loop unrolling (BLT, CADDY, Flamel, and HERCULES), and operator strength reduction ([Tutorial]). In addition, some systems attempt to transform the resulting data flow graph into a canonical form (DRT and [VSS]), removing stylistic programming idiosyncrasies. It should be noted that some of these optimizations and transformations are applied interactively (DRT).

These optimizations can be grouped into four categories ([CORALII]): data transforms (such as dead code elimination), control flow transforms (such as inline expansion), code motion (such as moving code into and out of case arms), and process transforms (creating processes and pipeline stages).

The Viper translator performs two of these optimizations, a data transform, dead code elimination, and a control flow transform, inline expansion. A third optimization, code motion, is performed by the trace scheduler described in Chapter 7.

The translator makes a separate pass over the completed register transfers and state transitions, and identifies unreachable states and their associated transfers. It also removes empty states that are only used for indirection, that is, transitions that simply move to another state.

It also expands all procedures inline. Without microsubroutines this is necessary. Furthermore, in the case of unconditional (single clause) procedures, which become part of the block into which they are expanded, it is desirable to do this because the more operations there are in a block the greater the opportunity for parallelism (an idea behind trace scheduling -- see Chapter 7).

There are four classes of procedures for inlining purposes:

- Those that have only one synthesizable clause. Such procedures simply become merged with the calling block. No separate basic block is created. The fetch procedure from Chapter 4, for example, is such a procedure.
- Those that have literals in clause heads. These procedures are effectively case statements. One basic block is created for each clause. The execute procedure from Chapter 4 is such a procedure.
- Those that have only ground clauses (no clause bodies). Such procedures are realized as tables (PLAs), and become a special type of register (input values are transformed into output values). No basic block is created.
- All other multi-clause procedures. These procedures are general conditionals, usually with a test in the body of each clause. One basic block is created for the test, and one for the rest of the clause body.

Also note that the formal parameters of Prolog procedures do not have modes. Any parameter on any call can be bound on entry (an input parameter) or unbound. This means that each call on a procedure can in effect invoke a different one, with different parameter modes. This requires a substantial amount of bookkeeping by the translator, which must monitor the binding status of variables and instantiate different procedure versions. The key to this bookkeeping is a naming convention described below.

3. Mapping Behavior to Hardware Function

An important issue in synthesizing hardware structure from behavior is determining which type of functional unit to use in implementing a given behavioral operator or expression. Synthesis systems sometimes assume a generic type of ALU and a generic register type. Generating realistic microprocessors, however, requires being more sophisticated in this mapping process, and using more specialized functional units.

There are a number of places in Viper where the system recognizes the opportunity for such optimizations. They resemble peephole optimizations, in that they are generally local in context. The contexts become more local as the synthesis process proceeds, moving from expressions to individual operators.

These optimizations can be implemented in two ways. The first method of implementation consists of performing the optimizations inline, by doing (clause head) pattern matching in the Viper code. This is how expression mapping is done (see below). The second method consists of consulting ancillary library files that, given some input, return an appropriate mapping. This is the way that most operator mapping is done.

These two techniques appear similar, but in practice are used for different problems. The first method is appropriate for mappings that require more context, such as those for expressions, or in circumstances where the problem is not well understood or easily parameterized. The second method is good in limited, well understood contexts.

The collection of ancillary mapping files used by Viper constitute the Viper functional unit library, and contain Prolog operator mappings and physical characterizations of functional units.

The two subsequent sections present specific mapping tasks.

3.1. Expression Mapping

The RTL translator produces register transfers as described above. These transfers are optimized in an associated stage (called mort -- MOdify RTs), that produces an equivalent collection of register transfers, with some transformed.

In particular, constructs such as adding and subtracting one, testing for one and zero, and setting fields to one and zero, are recognized and transformed. For example, the transfers

```
rtran(..., pc, constant(1), +, pc).
rtran(..., r1, constant(1), -, r2).
```

become

```
rtran(..., none, none, inc, pc).
rtran(..., r1, none, dec, r2).
```

The mort phase also keeps track of transfers that use temporaries, and makes lists of transfers that are part of the same expressions (and thus must be scheduled together). The expression above, for example, involving the addition and shift, would generate

```
exprt([10, 11]).
```

3.2. Operator Mapping

The mapping of a Prolog operator to hardware basically involves four pieces of information: the Prolog operator, the equivalent hardware function, the type of functional unit needed to implement the function, and the generic class to which that functional unit belongs. This information is kept in a library file (called opmap), and has the form

opMap(<op-name>, <fn-name>, <unit>, <class>).

In general, *<fn-name>* can be thought of as a renaming, *<unit>* as a binding, and *<class>* as a classification. The *<op-name>* *<fn-name>* pair is unique; no two appear in different opMaps. The *<fn-name>* alone is not unique; different units may implement the same function (such as increment). In such a case, the first opMap in the file with the *<fn-name>* is the default implementation (it is the first one found by Prolog). Also, associated with each class is the maximum number of buses it uses (in the structure opBusUse). This information could alternatively be specified for each unit type.

For example, the plus operator is mapped with the entry

```
opMap(+, add, alu(Type), arlog).
```

translating it to the add operation of the ALU. Note that the exact type of the ALU (ripple, carry bypass, etc.) is left unbound. It becomes bound later during the allocation phase. Also note that the ALU belongs to the class arlog, which also includes incrementers.

In addition to this table driven mapping, some operators are recognized specially, using unification pattern matching, before the general mapping is done. These special cases involve incrementers and counters, and setting bits to zero and one.

Operator mapping occurs during hardware allocation, which is described in Chapter 8. The complete opmap file can be found in [Viper].

4. Enabling Conditions

The register transfer translator also generates control dependency information later used for common case allocation (the subset feature described in Chapter 8) and pipelining (see Chapter 13). For each block, the translator produces a list of enabling conditions for that block. These conditions have the general form

blen(<block>, [<condition-list>]).

If a block, for instance, implements a specific instruction, the execution of that block depends on the fetched opcode being the opcode for that instruction. Consider, for example, the add instruction above. It is dependent on the opcode case dispatch (which is performed in register transfer 4), and is represented

```
blen(block(4), [case(rt(4), field(memDR,opcode), add)]).
```

This case structure indicates the register transfer in which the case dispatch is done, the value source, and the particular value needed. Other possible conditions are conditional tests (see cond below) and true (unconditional).

5. The Operation of the Translator

The RTL translator is the largest and most complex component in Viper. Details of its operation are presented in this section.

5.1. Procedures, Clauses, and Variables

Understanding the operation of the RTL translator requires understanding Prolog procedures, clauses, and variables, and the way they are designated by the translator. Remember that each goal, or procedure invocation, can have different arguments bound or unbound. This means that each goal can be thought of as calling a (potentially) different procedure. Thus, in a specification,

- a procedure's name is its identifying symbol and its arity,
- a procedure invocation consists of the procedure name and an invocation instance number,
- a clause name consists of a procedure invocation instance and a clause index number, and
- a variable name consists of a clause name and a variable index number.

Thus, for example, in

```
p(A), p(B), ...
p(0).
p(V) :- ...
```

the designation for V associated with the second invocation p(B) is variable 1 of clause 2 of invocation 2 of the procedure p/1.

The structure of the code that processes input specifications in general follows the Prolog clause structure; the Prolog *clause* built-in is used to traverse the input. Viper procedures exist for processing input procedures, their component clauses, and the elements of individual clauses -- clause heads and arguments and clause bodies. The different types of procedures -- simple inline, case, conditional, and table -- are identified, and their arguments and block structure are handled appropriately. Caller and callee variables are matched and modes (bound, unbound) are propagated.

5.2. Data Flow and Control Flow Graphs

The translator first generates data and control flow graphs from an input specification. These graphs are the highest level of abstraction after executable Prolog, involving only Prolog operators (no functional units) and no scheduling other than that required by dependencies.

Data flow nodes track the flow of data in the specification, and are of three types, source nodes, expression nodes, and destination nodes. Source nodes provide values from architected registers or constant values. Expression nodes perform computations on sourced values. Destination nodes store values in architected registers. The arcs between nodes represent the flow of values. Data flow nodes are represented in Prolog with the facts

```
flowSrc(<block>, <source>, <value-arc>).
flowExp(<block>, <source-value>,
        <source-value>,
        <operator>,
        <destination-value>).
flowDst(<block>, <destination>, <value-arc>)
flowArg(<block>, <type>, <old-arc>, <new-arc>).
    <type> ::= condin | condout | inline
```

Each value arc is given a unique name, based on the context in which it appears. In fact, each arc corresponds to a Prolog variable in the original specification, and takes on its associated variable name. The flowArg construct associates variables in different procedures that are bound together by procedure calls (goal invocations).

For example, the following data flow nodes produce a register transfer that increments the program counter.

```
flowSrc(block(2),pc,v(c(i(p(fetch,0),1),1),1)).
flowExp(block(2),
    v(c(i(p(fetch,0),1),1),1), 1, +,
        v(c(i(p(fetch,0),1),1),2)).
flowDst(block(2),v(c(i(p(fetch,0),1),1),2),pc).
rtran(rt(3),block(2),pc,constant(1),+,pc).
```

Control flow nodes divide the specification into regions (blocks) of control, and consist of two types, labels and jumps, which are represented with the Prolog facts

label(<context>, <tag>, <block>).
jump(<from-block>, <type>, <to-context>).
<tag> ::= <case-literal> / none
<type> ::= jump / cond / case

Labels associate clause names (see above), additionally annotated with literal tags in the case of case arms, with basic block identifiers. Jumps are, in general, transfers of control to named clauses, or to sets of clauses in the case of cond and case.

There are five types of jumps:

- Case dispatch. This jump selects and transfers to a case arm based on a tag value.
- Conditional test success. This jump simply transfers to the next block (the remaining goals in the clause).
- Conditional test failure. This jump transfers to the next clause.
- End of case or conditional clause. This jump transfers to the end of the entire case or conditional procedure (using a manufactured label).
- End of tail recursive clause. Such a clause has as its last goal an invocation of itself. The jump becomes a transfer to the first clause of the procedure.

5.3. Conversion to RTL

The data flow graph is next converted to the RTL (*rtran*) form presented above. Not that the operations performed do not involve functional units, but are instead abstract operators, namely the Prolog operators (built-ins) appearing in the data flow graph. This RTL translation process assigns associated data flow nodes to a specific basic block and an "instant" for scheduling purposes (although pipelining may stretch the instant out in a regular manner).

The control flow graph is simplified into finite state machine transitions, combining jumps and labels into block-based transitions. Each possible transition to a cond or case arm is defined with a separate transition fact. The facts have the form

stran(<from-block>, <condition>, <to-block>).
<condition> ::= true
/ case(<switch>, <tag>)
/ cond(<operator>, <source1>, <source2>)
/ cond(not(<operator>, <source1>, <source2>))

5.4. Error Detection

Detecting errors in input specifications is important to the robust operation of the translator, and is difficult in Prolog, because failure is not necessarily an error. Experience with error detection is discussed in [Viper], in the individual sections describing specific microprocessors. An additional register transfer scanning tool (rover) was constructed to aid in error detection. It identifies anomalies associated with specification errors.

Chapter Six: Common Case Optimization

The overall goal of this work is to apply the idea of common case optimization to high-level hardware synthesis. The first part of this chapter describes how common cases are identified in this synthesis context -- how addressing modes and instruction frequencies are determined and related to addressing mode and instruction definitions in specifications. The second part of the chapter discusses general common case optimization ideas. Subsequent chapters describe the common case optimizations used in Viper.

1. The Nature of Instruction Frequency Statistics

Instruction frequency statistics, the counts of how frequently the various instructions of an architecture are used (running a given set of programs), are a specific form of information that can more generally be called usage statistics. Usage statistics refer to the information that identifies how frequently the various parts of a design (of any hardware specification) are used. The following material is presented in terms of instruction frequencies, but the mechanisms described are not specific to microprocessors.

1.1. Simulation

Instruction frequencies are determined by two components:

- a hardware specification that defines available functionality (an instruction set architecture), and
- a collection of benchmark software that, in conjunction with the specification (when executed on the architecture), quantifies the use of the functionality.

The specification is, when simulated, an interpreted program, and the benchmarks its inputs, and the usage statistics are thus the result of instrumenting the program during execution (simulation). The specification is subsequently compiled (synthesized), using the usage statistics.

The usage statistics, being measurements of a program, consists of branch probabilities and block execution counts.

Viper collects this information at the register transfer level; branches and basic blocks are much easier to monitor here than at the Prolog specification level. Register transfers are not directly executable in Prolog. An RTL evaluator is used to interpret register transfers and finite state machine transitions.

A frequency count is maintained for each basic block and branch. This information has the form

freq(<tag>, <block>, <count>).
bran(<from-block>, <to-block>, <count>).

The tag argument in the frequency structure is effectively the name of the instruction the block implements (instruction dispatches are case statements; the tag is in fact the case tag associated with the block) and is an aid to human readability.

1.2. Annotation and Propagation

Not all usage statistics are gathered efficiently from direct simulation. Published instruction frequency counts, in particular, are useful, especially for large benchmarks and lengthy simulations. Also, the implementation of a processor may change, resulting in a changed specification with a different block structure, and yet the instruction frequency statistics may still be accurate. Here, also, it would be inefficient to resimulate.

Thus a second source of usage statistics, in addition to direct simulation, is annotation. The problem with annotation arises from the need to relate usage statistics to specific blocks, identified by internal names generated by Viper. The solution is to employ a separate binding phase which relates counts to blocks through the use of instruction names and their equivalent case tags.

The basic count information is represented as facts of the form

count(<tag>, <count>).

which are then bound to block names (retrieved from tagged state transitions), producing frequency structures (see above). (This assumes tags are unique.)

There is another, similar case where annotation is useful, but for which propagation and binding to block names is more complex. This case arises through scheduling optimization techniques that modify the specification and duplicate parts of it -- that produce duplicate implementations of instructions (see trace scheduling in the next chapter). Here the problem is to bind counts to instructions, with the counts modified by the probability of that particular instance of an instruction implementation being executed. This is again done by the binding phase, which binds execution probabilities, derived from the count data, to blocks, by propagating probabilities through the modified specification using its state transition graph.

The probability binding phase performs the following steps.

[1] It computes the probability of execution of each instruction and addressing mode from the count data. It does this by summing up the counts for instructions and addressing modes separately, and dividing each specific count by the appropriate sum.

This yields a probability for each tag, of the form

tprob(<tag>, <probability>).

In order to perform these computations the system must know the type of each count tag -- whether the tag identifies an instruction or an addressing mode. This information is packaged in structures that have the form

order(<class>, [<list-of-tags>]).

For example, the 6502 has

```
order(addressing, [zerop, ...]).
order(instructions, [lda, ...]).
```

Importantly, the tags in an order structure are also ordered by frequency. This ordering is important, and is used in optimization.

[2] It locates the cycles in the state transition graph. This is necessary for the correct operation of step three.

[3] It traverses the state transition graph, beginning in the start state with an execution probability of one. It labels each block (state) with the sum of the execution probabilities of its predecessor blocks, each multiplied by the transition probability computed in the first step. The result is an execution probability for each block, of the form

eprob(<block>, <probability>).

From these, frequency structures can be generated.

2. A Figure of Merit

Usage statistics can be used to calculate performance metrics for a design.

Cycles per instruction, a common processor metric, is defined in [Arch-H&P], page 37, as the sum, for all instructions, of the number of cycles needed to execute each instruction type times the probability of that instruction type being executed.

This metric translates, in the Viper context, to the sum, for all blocks, of the cycle length of each block multiplied by the probability of the block's execution; a block's cycle length, after scheduling, is the number of cycles needed to execute it. The Flamel system ([Flamel]) uses this metric to evaluate design choices, and the BUD ([BUD]) and Siemens ([DFBS]) clustering systems compute the average cycle time for a design in a similar fashion.

Viper also uses this metric. It is essentially a measure of delay -- the larger the number, the slower the design. The longer a block is and the more frequently it is executed, the greater its weight. Hence the smaller the metric the faster the design -- the shorter the more frequently executed blocks. Viper calculates this delay number for a design, using execution probabilities as computed above, and scheduling information described in Chapter 8.

This metric is used extensively in Chapters 10, 11, and 12 to evaluate the speed of synthesized designs.

3. Common Case Optimization: Counts, Probabilities, and Orderings

Human designers, in more or less formal ways, optimize common cases. Such common cases serve as cues as to which parts of the design are important. On the other hand, common cases cannot be given absolute precedence. Global delay (critical path) and area constraints must be met, and less common cases (less common instructions) usually must still be supported. Thus common case optimization is a greedy prioritizing technique in the context of global constraints. It is also, in a sense, a global technique, because usage statistics represent global information, indicating the overall relative priorities of various sections of a specification.

As can be seen from the above, usage information ultimately takes on several forms: raw counts, probabilities, and orderings. It can be used in several ways as well.

Counts and probabilities can establish thresholds, above which resources are allocated and below which they are not. In microprocessor synthesis such thresholds can be used to decide which instructions will be implemented. Probabilities are also used in performance metrics, as with Flamel and BUD.

Orderings can be used to control the processing of a design, with more important parts of the design processed earlier. Trace scheduling ([Trace]) is a microcode compaction technique that uses such orderings, and is adapted in Viper for hardware synthesis (see Chapter 7). The SUGAR system ([SUGAR]) also uses orderings, allowing the user to annotate manually a specification with ordering information, attaching priorities to the arms of case statements; Execution paths through the case arms are optimized in the order specified by the user, by moving operations into and out of the case arms in a manner similar to trace scheduling.

Chapter Seven: Common Case Scheduling

The goal of scheduling is to get the most done in the shortest possible time. In the context of hardware synthesis, this goal takes the form of maximizing concurrency within constraints. One constraint that affects most scheduling techniques is the condition that each basic block is a separate scheduling domain, and operations in one basic block cannot be moved to another. Such techniques cannot exploit interblock parallelism.

Trace scheduling is a well known technique that avoids this limitation, gaining higher degrees of concurrency by scheduling operations based primarily on data dependencies and moving operations over block boundaries ([Trace] and [Bulldog]). It was developed originally as a microcode optimization technique, but more recently has been applied to VLIW architectures.

It operates by scheduling different execution paths, or traces, through a specification in turn, scheduling the most frequently executed trace first. Scheduling a trace typically involves moving operations over block boundaries, which usually requires duplicating those operations on other traces (this movement process requires extensive bookkeeping). The result is that the more commonly executed traces are shorter and have more concurrency.

In general, trace scheduling trades control size and redundancy for concurrency and speed. It is essentially a greedy technique (greedy by trace), each trace being optimized globally within the constraints created by its predecessors.

Viper's common case scheduling phase uses a trace scheduler written in Prolog by Richard Carlson, another student in the Aquarius project [Trace-Carlson]. Viper translates its register transfers and state transitions into the form used by Carlson's scheduler (with conditional state transitions ordered by frequency), and then translates the results back into its own format. The resulting reordered transfers proceed as usual through synthesis.

Carlson's scheduler (referred to by him as a compactor) accepts optional constraints on the number of ALU operations, bus transfers, and memory reads and writes per cycle. Viper uses the scheduler without constraints, imposing them later during allocation. Carlson's notion of a single type of combinational unit, ALU, is too simple for effective allocation.

1. Scheduler Input Translation

Viper register transfers are translated into Carlson's form by scanning transfers block by block and inserting control information between blocks of transfers; Carlson's form has a block-like structured syntax, which makes it very human readable.

In addition, usage statistics are employed in ordering the transitions out of a state. The trace scheduler assumes the choices in a set of conditional state transitions are ordered by frequency, and uses that assumption in selecting traces to schedule. Viper generates the transitions accordingly, using the order structures described in Chapter 6.

Input to the scheduler has the form

```
<specification> ::= [ <operation> {, <operation>}, end].
<operation> ::=
    <destination> <-- <source> /
    label( <tag> ) /
    goto( [ <target> {, <target>} ] )
<target> ::= <tag> / ( <test>, <tag> )
<test> ::= ( <value> <comparison> <value> )
<value> ::= <register> / <constant>
```

Sources and destinations can be registers, fields, tables, and memory locations (indicated by mem(memAR)). In addition, sources can be constants and expressions. For example, the add instruction presented in Chapter 4 would be translated to

```
label(block(4)),
    memAR <-- field(memDR,address),
    memDR <-- mem(memAR),
    ac <-- memDR + ac,
  goto([block(2)])...
```

In addition, the scheduler supports constraints that force different transfers to be scheduled on the same cycle. Viper uses this feature to force expressions using temporaries to be scheduled together. For example, the following constraints force the transfers, computation and test, to be done in the same cycle (otherwise the temporary might have to be a latch).

```
{[], [], [force(m1,0), force(m2,0)]},
    temp(1) <-- acc + memDR,
{m1},
    acc <-- temp(1),
{m2},
    field(p,zflag) <-- 0 = temp(1)...
```

(The other lists and the zeros are for other types of constraints, unused by Viper.)

Unfortunately the above mechanism is not entirely robust, and an alternate mechanism is also used. This mechanism supports a list of destinations, all of which are scheduled together. In this form, temporaries are implicit, as are the operations on them (such as the zero test above). For example, the above addition can be specified

```
[acc, field(p,zflag,isZero))] <-- acc + memDR...
```

The third, isZero argument in the field structure indicates that the field should be assigned the value of a zero test done on the result of the addition.

2. Scheduler Output Translation

The scheduler generates output that has almost the same form as its input. The output differs in that the transfers are grouped in lists, and resource usage numbers are supplied in braces. For example, the output version of the simple addition instruction above is

```
[ label(from1to2), label(safe2), memAR<--field(memDR,address),
  goto([from2to1]), {1,1,0,0} ],
[ label(from2to1), memDR<--mem(memAR), {0,1,1,0} ],
[ ac<--memDR+ac, goto([safe1]), {1,1,0,0} ],
...
```

The resource usage monitors (unused by Viper) are number of ALU operations, buses used, memory reads, and memory writes.

The trace scheduler output is then converted back into Viper form, by

(1) translating the register transfers into Viper format,

(2) creating block names, labels, and gotos,

(3) creating state transitions from the labels and gotos,

(4) eliminating dead code, and

(5) generating state transitions with cases and conds in Viper format.

The result is a file that is exactly the same in format as the output of the Viper Prolog to RTL translator.

Chapter Eight: Common Case Allocation

This chapter describes the Viper data path allocator. It first discusses common case allocation, and highlights other important features of the allocator. It then details the step by step operation of the allocator.

1. Highlights of Allocation

This section covers the important features of the Viper allocator. It describes the constraints the allocator uses, how it selects functional units, and how it performs common case allocation. In general, the allocator was modeled on the MAHA allocator ([MAHA], see Chapter 2).

1.1. Global Constraints

Viper allocation is controlled by a delay constraint and a total area constraint, both supplied by the user. (There is an additional constraint, the maximum number of buses possible, that currently comes from the technology.)

The delay constraint, essentially a clock speed constraint, is given precedence in Viper, as it often is in microprocessor design. Such designs are usually driven by memory system speeds, which bound clock speeds.

The delay bound is also a more local bound than the area bound. A given functional unit's delay contributes only to the delays on the paths that unit is part of, while the unit's area affects the total chip area. It is possible to partition total area into distinct clumps dedicated to different classes of functional units ([Chippe]), as a way of localizing area contributions, but Viper does not do this.

Viper uses the delay constraint to select individual functional units, and uses the area constraint to control the total number of functional units allocated.

The constraints are defined in an auxiliary file, and have the form

max(delay, <time>). % nanoseconds
max(area, <space>). % square microns

In addition, for design end points, maximally serial designs (no duplicate units) and maximally parallel designs (as many duplicates as required by as-soon-as-possible scheduling) can be generated automatically by specifying special serial and parallel constraints. These constraints override the standard global area constraint, and have the form

max(ser).
max(par).

1.2. Functional Unit Classes

There are in general two classes of functional units for allocation purposes, those necessary to implement a design, and those extra units that will increase the design's performance.

The first class includes the required set of architected registers, and one each of the combinational logic units needed to perform the computations required by the specification, such as ALUs and shifters. These units are just those needed for a minimum area, low performance serial design (that is, where register transfers are evaluated serially, with no concurrency among transfers).

The second class consists of those additional units which, when added to a serial design, provide duplicate functions and allow operations to be done in parallel. This class includes such items as extra ALUs and shifters, counter registers, and additional buses for transferring data in parallel.

If, given an area constraint, a design cannot be created with the required functional units, it fails. On the other hand, once the required functional units have been allocated, the optional functional units simply improve the performance of the design.

Common case allocation orders the selection of the functional units in that second, additional class. It provides a metric for selection. The most frequently used optional units are allocated first.

1.3. Functional Unit Modification

Sometimes the functional unit implementing an RTL operator can be modified and optimized. There are two circumstances when this can occur.

The first arises because, in mapping an operator to a functional unit, the system chooses the most general applicable unit. This overall strategy is sensible because, by and large, the more general a unit is the more widely it can be used.

Sometimes, though, a unit can be specialized. In such a case, a functional unit is degraded, from, for example, an ALU to an incrementer, or from a barrel shifter to a one bit shifter, or from a comparator to a zero detector.

The second unit changing circumstance arises because the system initially implements registers as simple load-store storage elements without combinational functions.

Occasionally, however, a register can be augmented with a combinational function. This may, for example, occur with the register containing the program counter; it could be changed to a counter, so that program counter operations could take place in parallel with other arithmetic operations without requiring a complete second ALU and associated buses.

1.4. Common Case Binding

The process of associating the operators in individual register transfers with hardware functions is a definitive act of synthesis. It implies a register transfer schedule and an allocated set of functional units.

This binding process takes place in Viper after a tentative, as-soon-as-possible register transfer schedule has been calculated and functional units have been allocated through global needs analysis, this allocation being limited to units that are clearly required. The binding process may reveal that, in any cycle, there may not be enough hardware resources to perform all scheduled operations (because of concurrency). There are two possible solutions to this problem, allocate more resources, or delay operations. Viper employs both these solutions, ordering them in an attempt to produce a good overall design.

Viper first attempts to add functional units as needed, to support maximum concurrency. This attempt may fail, because the requisite unit may exceed the total area bound for the design. If it fails, transfers will be delayed until resources are available. This delay may not be significant. It only affects the ultimate speed of the design if the transfer and its dependents are among those that determine the total number of cycles in the block (the block's critical path). Consider, for example, the fetch clause from Chapter 4, and its associated register transfers.

```
access(pc, PC), set(memAR, PC),
mem_read,
access(pc, OldPC), NewPC is OldPC+1, set(pc, NewPC).
rtran(1, block(2), pc, none, move, memAR).
rtran(2, block(2), memAR, none, mem_read, memDR).
rtran(3, block(2), pc, none, inc, pc).
```

Transfer 2 is dependent on transfer 1; these two transfers require that the block be two cycles long. Transfer 3, on the other hand, is dependent on neither, and is not part of the critical path. It may be delayed from cycle 1 to cycle 2 without affecting speed.

Binding in Viper is essentially a local activity, performed block by block, cycle by cycle, and transfer by transfer. The order that blocks and transfers are processed is important.

Viper processes blocks in order of decreasing frequency. This guarantees that the extra resources allocated to support concurrency will be used in the more important blocks. These resources may or may not be useful in the less important blocks. In contrast, less frequently executed blocks are more likely to be delayed. Such delays are less important to the overall performance of the design.

Viper also orders the processing of transfers in a cycle, basing the ordering on the scheduling freedom, or mobility, of each transfer. Consider the above fetch example. The block requires two cycles to execute. Transfer 1 must execute in cycle 1 and transfer 2 in cycle 2. Transfer 3 can execute in either cycle 1 or cycle 2. Transfer 3 is thus mobile, while transfers 1 and 2 are not. This concept (freedom in [MAHA], mobility in [Chippe-Micro]) refers to the number of potential cycles a transfer can occupy; it is the difference between a transfer's as-late-as-possible and as-soon-as-possible cycle assignments. Transfer 3 has a mobility of 2, while transfers 1 and 2 have mobilities of 1.

Viper processes transfers in increasing order of mobility, binding the less mobile transfers first. Thus scarce resources are bound to less mobile transfers, with the expectation that more mobile transfers will be harmlessly delayed and bound in a later cycle. The goal is to reduce the effect of delays on performance.

1.5. Individual Functional Unit Selection

Viper selects each functional unit in turn using the delay constraint. For a given type of functional unit, such as ALU, the library of functional units is searched, from the slowest (and smallest) subtype (such as ripple carry) to the fastest (and largest) subtype (such as carry select). The first functional unit that meets the delay constraint is chosen. Thus each selected functional unit is the smallest that meets the delay constraint.

Currently, Viper simply applies the delay constraint directly to each functional unit selected. This is a useful approximation because expressions in the specifications encountered so far are simple, and a source-operator-destination model is not grossly inaccurate. A simple and easy improvement would be to add source and destination delays. Another simple improvement would be, in an expression with more than one operator, to consider all subtype combinations of operators, order them by total area, and choose the first (smallest) combination that was less than the given delay constraint. This would not be that expensive because, again, expressions are simple.

Note that Viper uses the delay bound only for the data path, not for the control path.

Once the smallest functional unit meeting the delay constraint has been found, its area is added to the total area consumed by the design and that total is checked against the global area constraint. If this check fails the design is unacceptable.

1.6. Characteristics of Functional Units

Two sets of functional unit libraries are available to Viper. The first is the ASP set, used by the ASP system. The second is the Harvest set, developed as part of the Harvest CAD system.

The Harvest CAD system, developed by Harvest VLSI, is a sticks-based layout system, similar to the lower levels of ASP, but written in C. It takes as its input structural design descriptions similar to those generated by Viper, but, unlike ASP, uses human guidance to produce compacted layout. The Harvest library is well parameterized and complete, at least in terms of microprocessor synthesis. Its elements consist of compacted cells that have been characterized with SPICE, and thus its physical properties are well defined. It has been used to generate an implementation of the BAM microprocessor. It can be found in [Viper].

For Viper, a functional unit has three basic characteristics, a type, a size, and a delay. For an element in the ASP library, the functionality of which is specified in terms of gates, its size is the total number of gates, and its delay is the longest path (in number of gates). For an element in the Harvest library, which consists of compacted cells, its size is in square microns and its delay is in nanoseconds.

For example, the ripple carry Harvest ALU is characterized by

```
type(alu(ripple)).
delay(alu(ripple), N, D) :-
    D is 2.5 + (1.2 * N), !.
size(alu(ripple), 195, 45).
area(Type, N, A) :-
    size(Type, X, Y),
    A is X * Y * N, !.
```

The *delay* and *area* clauses take as input the width in bits of the ALU (N), and produce an appropriately scaled result.

The functional unit selection process uses Prolog's search and failure mechanism, which relies on the order of elements in the functional unit library. Small, slower units appear first.

1.7. Generating Design Subsets with Utility Constraints

Common case allocation as described thus far automatically limits resources based on area and delay constraints. It can also limit resources based on a utility constraint. That is, it can ignore the parts of a specification that are utilized below some threshold amount, and simply not implement them.

The obvious example case is that of unused instructions. If a given application does not use an instruction, and the processor is to be specialized for that application, then the implementation of that instruction can be omitted.

It can also be desirable to omit seldom-used instructions, saving on data path and control area (and possibly increase clock speed), and implement them instead in software. Some instructions, however, may be important although seldom used (such as return from interrupt).

Viper supports all these case by using an optional utility constraint. When given, the allocator simply does not allocate functional units used by instructions that occur less frequently than the given value, effectively leaving such instructions unimplemented in hardware. The constraint has the form

max(min, <bound>).

In addition, implementation of instructions can be forced despite their numerical utility, through use of the declaration

keep(<instruction>).

Thus the implementation of all instructions used less that 1%, with the exception of return from interrupt, is suppressed with

```
max(min, 0.01).
keep(rti).
```

2. The Operation of the Data Path Allocator

The data path allocator is composed of four modules, sched (which performs initial as-soon-as-possible and as-late-as-possible scheduling), needs (which determines overall hardware needs), decl (which actually creates and binds functional units), and nets (which creates and schedules bus operations).

The specific steps the allocator takes are:

[1] **Compute a tentative block-based as-soon-as-possible schedule (sched).** Data dependencies between transfers are determined and transfers are scheduled on a block by block basis, with each transfer assigned a cycle number within its block. To aid in allocation (for computing mobility), the as-late-as-possible schedule for each block is also computed.

The cycle assignments are represented as

cycle(<index>, <block>, <cycle>).

For example, the schedule for the three transfer memory read and add above, assuming they were the first three transfers in the block, would be

```
cycle(5, block(4), 1).
cycle(6, block(4), 2).
cycle(7, block(4), 3).
```

The scheduler keeps track of the current value of each resource (definition and use -- last write and subsequent reads) in each block. This bookkeeping is enough for dependency analysis and scheduling. The only complexity is that, for fields in registers, dependency information must be managed both for the field and the entire register (a field and the entire register cannot both be stored into during the same cycle). The value of a register is defined by the last write of the entire register or the last write of any field, whichever is latest; in contrast, the value of a field is defined by the last write of the entire register or the last write of that field.

The scheduler currently makes three simplifying assumptions, which are not restrictive given current inputs; the scheduler could be extended if the need arose. The assumptions are:

- different fields of the same register are non-overlapping (this simplifies dependency analysis);
- if a value is stored in a register, or retrieved from memory, then it will be available the next cycle (delays could easily be put in the last use retriever); and
- all expressions can be computed in one cycle (this assumes expressions are implemented as purely combinational logic; an expression is a collection of register transfers bound together in an exprt list -- see Chapter 5). In other words, register transfers using the same temporaries are codependent and are scheduled together.

To aid in allocation, the scheduler also computes the as-late-as-possible schedule for each block. The as-late-as-possible schedule is represented in a manner similar to the as-soon-as-possible schedule, and for the above example would be

```
aslate(7, block(4), 3).
aslate(6, block(4), 2).
aslate(5, block(4), 1).
```

[2] **For each register transfer, note the resources it uses, along with associated usage statistics (needs).** The allocator accumulates information on hardware needs, examining the operators and operands of each transfer in the design. The information on required elements is collected in structures of the form

necessary(<type>, <name>, <count>).
required(<frequency>, <count>, <type>, <function>).
required(<frequency>, <count>, <type>).

The *necessary* structures are for operands, usually registers. The *required* structures, accumulated for both specific functions (such as add) and general functional units (such as ALU), are for operators, and describe combinational logic. Information on specific functions is collected for the purpose of functional unit modification.

Note that the usage information appears first in required structures because Prolog's sorting operations (included in the setof operator) can be used to order sets of structures. The counts are simple unweighted occurrence counts, unused by Viper but maintained as a possible aid to the designer.

For example, the transfers and associated statistics

```
rtran(7, block(4), memDR, ac, +, ac).
freq(add, block(4), 0.15).
rtran(10, block(5), memDR, ac, Λ, ac).
freq(and, block(5), 0.05).
```

produce

```
necessary(reg, reg(ac), 4).
necessary(reg, reg(memDR), 2).
required(0.15, 1, alu(Type), add).
required(0.05, 1, alu(Type), and).
required(0.20, 2, alu(Type)).
```

Note that register names are packaged in the reg(...) structure, which makes it easy to identify such names as registers.

In addition, optional facts record alternative implementations of functions (using, for example, incrementers and decrementers).

optional(<optional-function>, <required-function>,
<block>, <cycle>, <option-type>).
used(<type>, <block>, <cycle>, <count>, <RT-index-list>).

This step also implements the optional design subset feature, ignoring the needs of blocks used less than a specified threshold amount (which are also not protected by a keep statement). Such blocks are labeled unused.

[3] **Compute functional unit summary use information (needs).** The usage information for each type of functional unit is summed.

[4] **For each special functional unit type get the set of required functions; if the set is restricted then modify the type (needs).**

This step implements functional unit modification, discussed above. The functional unit types are ALU, shifter, and comparator; the function sets are increment/decrement, one bit shift, and compare with zero; The new types are incrementer, one bit shifter, and zero detector.

[5] **For each necessary or required element, create it within area and delay constraints (decl).**

This step implements the functional unit selection process described above; the allocator creates a basic data path using its smallest first strategy.

Declarations have the form

elem(<name>, <type>, <delay>, <area>).
elemFn(<name>, <function>).

The delay and area information is not used further by Viper, but is included as an aid to the designer.

For example, the above add and and register transfers generate

```
elem(o(alu,1), alu(ripple), 21.7, 140400).
elem(reg(memDR), reg, 3.1, 59200).
elem(reg(ac), reg, 3.1, 59200).
elemFn(o(alu,1), and).
elemFn(o(alu,1), add).
```

This example is based on the Harvest library, hence the nanosecond and micron data.

Note the naming conventions for functional units. Combinational units are named using an o(...) structure (o for operator), with the structure arguments being the unit type and the unit number.

[6] **For each block, in order of frequency, and for each cycle within each block, bind register transfer operators to functional units; create new functional units and delay operators as necessary (decl).**This step implements the binding process described above, using frequency and mobility/freedom to order binding. Mobility is computed using the as-late-as-possible schedule generated during as-soon-as-possible scheduling, being simply the difference between the two schedules.

The results of this step, in addition to possibly more functional units, are bindings of functional unit functions to specific register transfers. These bindings have the form

enable(<name>, <function>, <block>, <cycle>, <transfer-index>).

which includes hardware and scheduling information. For example, the bindings of the add and and transfers above are represented as

```
enable(o(alu,1), add, block(4), 3, 7).
enable(o(alu,1), and, block(5), 3, 10).
```

given that they both use ALU 1 and both occur on cycle 3 of their respective blocks.

It may be the case that, due to lack of resources, an operator must be delayed beyond its range of mobility (in other words, the necessary resources are not available within its range of mobility). In such a case, a forced cycle assignment is generated. Such an assignment requires a return to step one, because the dependency-based as-soon-as-possible schedule created in that step is no longer valid.

Such forced cycle assignments have the form

forsch(<index>, <block>, <cycle>).

These assignments are recognized and used by the scheduler in step one.

This step also participates in the design subset feature, ignoring unused blocks as labeled in step two.

[7] **For each block and cycle, get the set of all register transfer source-destination pairs, and create and schedule buses to move data between those pairs (nets).**

This step creates buses between registers and functional units, and schedules bus use. It attempts to minimize the buses and connections created, within a greedy framework. It operates in five stages.

- It schedules all source-destination pairs which are connected by a free bus.
- It schedules all pairs in which the source is connected to a free bus. It connects the destination to that bus.
- It schedules all pairs in which the destination is connected to a free bus. It connects the source to that bus.
- For each remaining free bus it takes a remaining pair, connects the source and destination, and schedules the bus.
- For all remaining pairs it allocates a new bus, connects the source and destination, and schedules the bus.

The results of this step are scheduled data movements. These movements have the form

move(<bus>, <source>, <destination>, <block>, <cycle>).

For example, the moves of the add and and transfers above are represented as

```
move(bus(1), ac, port(o(alu,1),2), block(4), 3).
move(bus(2), memDR, port(o(alu,1),1), block(4), 3).
move(bus(3), o(alu,1), ac, block(4), 3).
move(bus(1), ac, port(o(alu,1),2), block(5), 3).
move(bus(2), memDR, port(o(alu,1),1), block(5), 3).
move(bus(3), o(alu,1), ac, block(5), 3).
```

Note the ALU input ports. The ports are needed because the ALU has two inputs that must be distinguished.

3. Building Data and Control Paths

A bookkeeping phase follows data path construction. It collects together the relevant facts about the developing design that have been generated during the previous phases and packages them into the finite state machine descriptions and data path netlists (described in Appendix D). It uses register transfers, state transitions, schedules, and functional unit and bus declarations and bindings to construct its output.

Chapter Nine: Experiment Overview

This chapter describes the nature of the experiments used to test the effectiveness of the optimizations described in previous chapters. It also describes the operation of the complete Viper system.

1. The Experiments

The experiments constructed to evaluate common case optimization used four variables:

(a) *Speed and area constraints*. Different speed constraints were used to control the allocation process. Maximally serial and maximally parallel designs were generated as end points. These constraints are described in Chapter 8.

(b) *Instruction Frequency Statistics*. Different sets of statistics, corresponding to different benchmark applications, were used with the same microprocessor specifications to synthesize different microprocessors. The benchmarks used are presented below.

(c) *Synthesis Paths*. The synthesis path through the Viper system was varied to optionally include common case scheduling and allocation. The various synthesis paths are discussed below.

(d) *Microprocessor Specifications*. Different microprocessor specifications of varying complexity were used. The four primary specifications were: a simple eight instruction microprocessor (the SM1), an 11 instruction subset of the 6502, the full 6502, and a contemporary general purpose microprocessor minimally extended to support Prolog (the BAM). These specifications are more fully described in subsequent chapters.

The results of synthesis were quantified, and the quantities measured were:

(i) the number of register transfers produced by the design, a measure of input specification complexity;

(ii) the number of basic blocks and execution cycles generated by the design, a measure of control size;

(iii) the size of the generated data path;

(iv) and the overall performance (CPI).

The quantified results are presented in subsequent chapters.

2. Benchmarks

The Systems Performance Evaluation Cooperative, an independent organization, maintains a standardized set of real world, applications oriented benchmarks ([SPEC]). The benchmarks are all large, long running (5-10 minutes), UNIX based programs. Three of the SPEC benchmarks were used in the experiments:

- *GNU C compiler (gcc)*. This benchmark consists of the compilation of 19 preprocessed source files into optimized assembly language.
- *eqntott*. This integer intensive test written in C translates a logical representation of a boolean equation into a truth table. It is 95% sorting.
- *spice 2g6*. This is an analog circuit simulation and analysis application, written in FORTRAN.

These are all substantial benchmarks. None of them were actually run using the Prolog specifications synthesized in the experiments. They were instead run on a simulator written by Harvest in C for another architecture (SPARC). This approach was taken for four reasons:

- Properly simulating the benchmarks would require properly targeted compilers, under UNIX operating system support, practically beyond the scope of this research.
- The benchmarks are quite large, and simulation speed is an issue. The gcc benchmark executes 1,241,283,108 instructions, the eqntott benchmark 1,469,792,037 instructions, and the spice benchmark

22,843,194,799 instructions. The difference in efficiency between Prolog and C is significant at this scale.

- With the annotation mechanism described in Chapter 6, actual simulation by the synthesized specifications is unnecessary.
- The goal of the research is to explore the efficacy of common case optimization. Precise benchmark data is in fact not necessary to test the optimization techniques. A variety of different instruction frequencies, to stimulate different optimization results, is the fundamental requirement of the benchmark data.

The Harvest SPEC results thus formed the basis of sets of instruction frequencies used to drive synthesis. The Harvest numbers were mapped to SM1 and 6502 instructions based on approximate similarities in functionality. These approximate mappings are at least as realistic as small directly simulated benchmarks would have been. The exact mappings are described in Appendix B and in [Viper].

In addition to the SPEC benchmarks, for some Prolog-oriented experiments a composite of 25 Prolog benchmarks was used, which executed in total 50,283,248 instructions (see [Suite]). These benchmarks included the eight Warren benchmarks (163,237 instructions) and two large benchmarks, an extract from a Boyer-Moore theorem prover (22,211,221 instructions), and a data base construction and querying program (20,256,394 instructions). The BAM Prolog compiler and simulator were used to compile and execute the benchmarks (see [BAM]).

3. Synthesis Paths

The Viper system was constructed to allow the optional use of components that could perform common case optimizations. In this way unoptimized designs could be compared with optimized ones. Figure 9-1 presents an overview of these different paths:

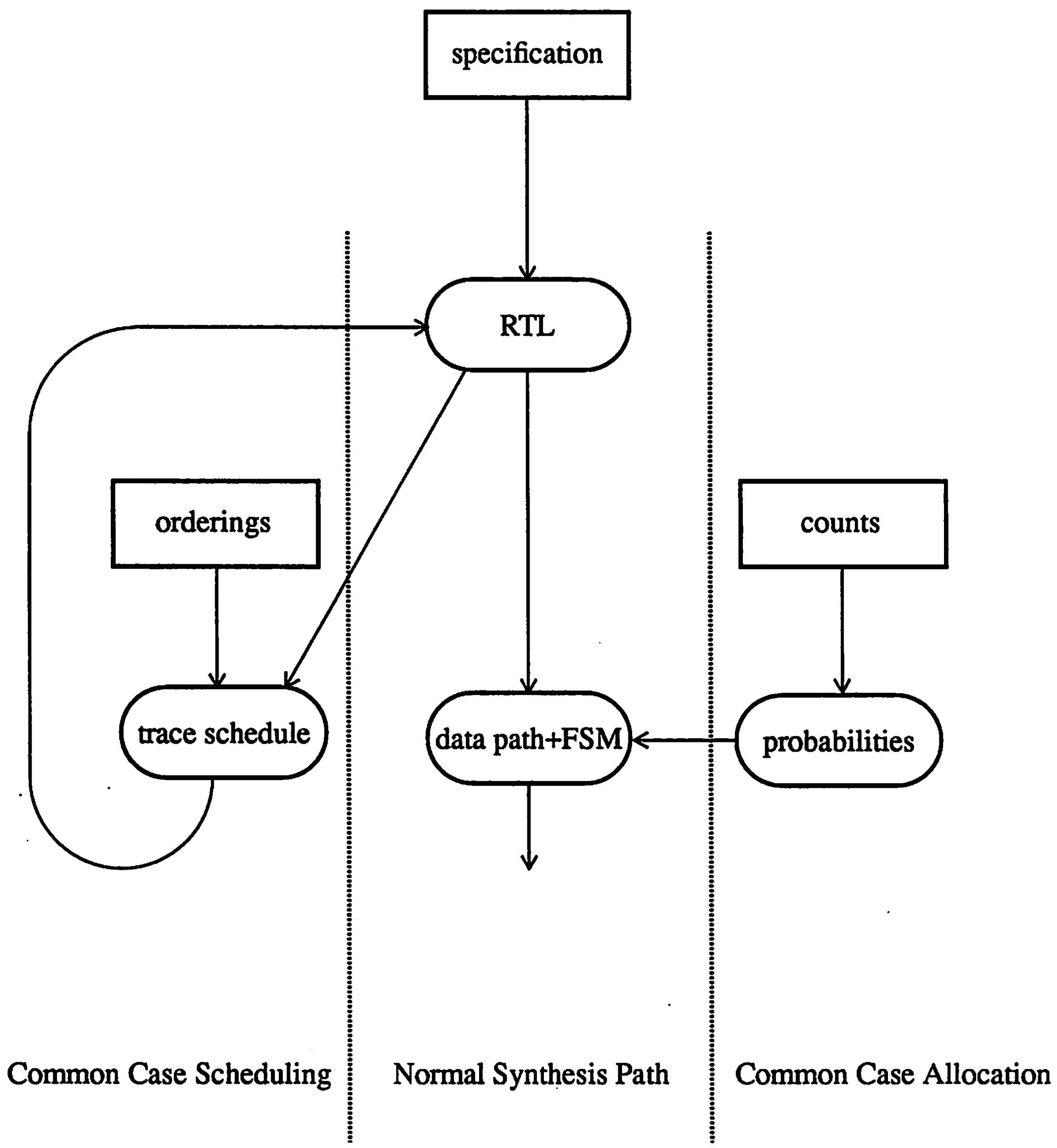


Figure 9-1: Synthesis Paths Through Viper

(1) The normal synthesis path translates a specification into register transfers, and then uses the register transfers to generate a data path and finite state machine.

(2) Common case scheduling transformations add an extra loop, taking register transfers and lists of instructions ordered by frequency of use, and generating new register transfers that are fed back into the normal path.

(3) Common case functional unit allocation uses instruction frequency counts to compute usage probabilities, which are in turn supplied to the Viper components that generate data paths and finite state machines. These components are constructed to use default, uniform values if probabilities are unavailable.

These paths are detailed more completely in Figure 9-2, which shows how intermediate representations (in the form of files) flow through specific Viper modules.

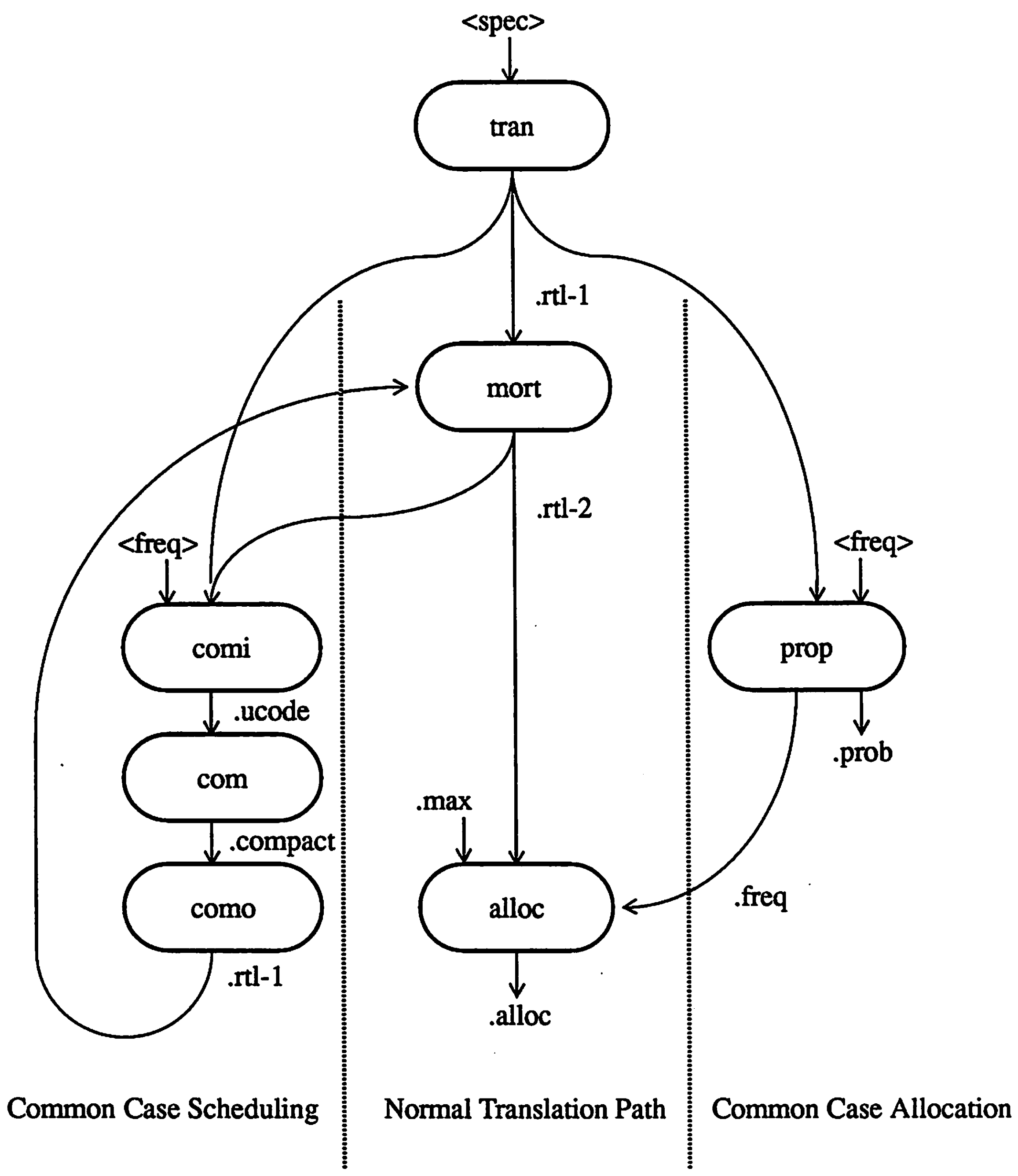


Figure 9-2: Information Flow Through Viper

The individual modules of Viper are:

- *tran* (see Chapter 5), which translates Prolog into register transfers;
- *mort* (see Chapter 5), which performs some simple optimizations on those transfers;
- *alloc* (see Chapter 8), which creates data paths and finite state machines (and consists of the component modules sched, needs, decl, and nets);
- *prop* (see Chapter 6), which computes probabilities from instruction counts, and binds and propagates them to specifications;
- *comi* (see Chapter 7), which converts Viper register transfers into compactor register transfers;
- *com* (see Chapter 7), the compactor, which performs trace scheduling; and
- *como* (see Chapter 7), which converts compactor register transfers back into Viper register transfers.

The input files for a design (for each experiment) are: the specification file (<spec>), the benchmark data file (<freq>), and the constraint file (.max). The names of the files identified with angle brackets can be arbitrary; the other files have as names the input specification file name prepended to the given extension.

Note that the common case scheduler (com) produces register transfers just like those generated by the translator (tran). In order to distinguish the two (so that files do not overwrite each other), the scheduler files are given a special t extension (thus sm1.rtl is the output of the translator, while sm1t.rtl is the output of the trace scheduler).

4. Using Viper

The Viper design process is managed with an executive, a Prolog program that the user loads and uses to invoke individual tools. The executive in turn loads the specific tool module and input files, and writes the output file with the proper extension.

Usage information (the count and ordering data of Chapter 6) is kept in a collection of files. These files are named by design type (such as 6502), and benchmark (such as gcc). When a tool is invoked that requires such a file, and more than one file is available, the executive queries the user for the specific data set name. Each such file also includes a fact of the form

dataSource(<source>).

These facts are propagated through the Viper components, so that designs contain a record of the information used to optimize them.

There are a few other modules that are part of Viper. There is a performance evaluator (thev), which computes the cycles per instruction metric described in Chapter 6, a block and cycle counter (bloc), a verification module (rover), and a latch insertion module (red) discussed in Chapter 12.

Both C-Prolog and SICStus Prolog are used. C-Prolog is better for development (because it does not require declarations); SICStus is better for larger problems (because it has garbage collection and can use more memory). One module runs only in SICStus (com); several modules run in both (prop, sched, needs, decl).

5. Functional Unit Library Summary

Viper functional unit library information has three different aspects, each of which is contained in a different file, and the total of which conceptually constitutes the library. The individual aspects have been discussed separately in prior chapters, and are summarized here:

- the mapping of Prolog operators to equivalent functional unit functions (contained in the opmap file);
- area and delay estimates for functional units (contained in the lib file); and
- implementations of functional units (defined in lower level ASP and Harvest files).

In addition, some mapping (mostly optimization) is done inline by code rather than via library table lookup (see the discussion of mapping in Chapter 5).

The complete opmap and lib files can be found in [Viper]; the process of adding elements to the library is described in Appendix C.

6. Behavioral Simulation Summary

Simulation was discussed in part in Chapters 4 and 6, and is summarized here. Viper currently supports two levels of simulation, one at the executable Prolog level, and one at the register transfer level. The executable Prolog level is good for debugging specifications, verifying that specifications are correct. The register transfer level is good for gathering instruction frequency statistics. The former level is supported by Prolog built-ins; the latter level by an RTL interpreter. Simulation at additional levels could be added (such as at the output of Viper), to verify the synthesis process and get more accurate electrical performance information.

Chapter Ten: Simple Machine Results

This chapter describes the Simple Machine 1 (SM1), parts of which appeared in previous chapters as examples. It then discusses synthesis experiments using the SM1, and presents the synthesis results.

1. Simple Machine Specifications

The Simple Machine 1 (SM1) is indeed simple. It has a single accumulator, a single addressing mode, and implements the eight instructions load, store, add, and, shift right one bit, jump, branch if negative, and halt.

This simple example has pragmatically proved quite valuable, because it is simple enough to be used by prototypes tools, and yet complex enough to be interesting and expose substantial issues.

It is presented here to complete and unify the examples given in previous chapters, and to show the simplest complete input given to Viper.

1.1. The SM1 Specification

The specification is composed of four main components: register declarations, a main tail recursive loop, an instruction fetch procedure, and a collection of execute clauses.

```
stateRegister(ac, 16).
stateRegister(pc, 16).
stateRegister(memAR, 16).
stateRegister(memDR, 16).
stateField(memDR, inst, opcode, 1).
stateField(memDR, inst, address, 2).
```

All of the registers are 16 bits wide. The opcode is the first field in the memory data register, and the address field is the second.

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

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

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.
```

These run, fetch, and execute clauses were discussed in Chapter 4. Only the add execute clause appeared in Chapter 4; all execute clauses are presented here.

Simulation of this specification requires the built-ins stateRegister, stateField, stateUpdate, access, set, mem_read, and mem_write, discussed in Chapter 4.

1.2. The Register Transfer Version of the SM1

The register transfer version of the SM1, generated by Viper and used for hardware allocation and scheduling, is similar in content to the input specification, but differs in the method of specifying computation and control. Separate access, is, and set goals have been associated into unified transfers. Prolog clause structure has been replaced by labels and gotos.

This version is presented here to illustrate complete input to the Viper scheduler and allocator; it is in the form accepted by the trace scheduler (see Chapter 7).

```
% Ordering data from spec
[label(block(2)),
    memAR <-- pc,
    memDR <-- mem(memAR),
    pc <-- pc +1,
 goto([
        ((field(memDR,opcode) = add), block(4)),
        ((field(memDR,opcode) = load), block(7)),
        ((field(memDR,opcode) = and), block(5)),
        ((field(memDR,opcode) = stor), block(8)),
        ((field(memDR,opcode) = brn), block(10)),
        ((field(memDR,opcode) = shr), block(6)),
        ((field(memDR,opcode) = jump), block(9)),
        ((field(memDR,opcode) = halt), stop)
  ]),
label(block(4)),
    memAR <-- field(memDR,address),
    memDR <-- mem(memAR),
    ac <-- memDR + ac,
 goto([block(2)]),
label(block(5)),
    memAR <-- field(memDR,address),
    memDR <-- mem(memAR),
    ac <-- memDR Λ ac,
 goto([block(2)]),
label(block(6)),
    ac <-- ac >>1,
 goto([block(2)]),
label(block(7)),
     memAR <-- field(memDR,address),
     memDR <-- mem(memAR),
     ac <-- memDR,
  goto([block(2)]),
label(block(8)),
     memAR <-- field(memDR,address),
     memDR <-- ac,
     mem(memAR) <-- memDR,
  goto([block(2)]),
label(block(9)),
     pc <-- field(memDR,address),
  goto([block(2)]),
label(block(10)),
  goto([((ac < 0), block(11)), block(2)]),
label(block(11)),
     pc <-- field(memDR,address),
  goto([block(2)]),
label(stop), end].
```

The other important inputs to the system are the instruction count data and the global design constraints. An artificial set of instruction frequencies, used in the simulation experiments, along with a common set of constraints, appear below.

```
dataSource(spec).
order(instructions, [add, load, and, stor, brn, shr, jump, halt]).
srcClass(field(memDR,opcode), instructions).
% 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).
max(delay,100).
max(area,10000000).
```

All of the SM1 intermediate forms generated during synthesis, along with the associated interaction with Viper, are presented in [Viper].

1.3. Other Simple Machines

Other versions of the SM1 appear in Appendix A. They include a multiport memory version, a pipelined version, and the SM2. The pipelined version is discussed in Chapter 13.

The SM2 is an enhanced version of the SM1, with two general purpose registers and an IO register. It has four additional instructions: subtract, register to register move, no-op, and branch on tag. Although simple, it is complex enough to support Aquarius Prolog. It was used in one synthesis experiment below.

2. Simple Machine Experiments

The SM1 was synthesized a number of times using different constraints and benchmark inputs.

(a) To determine design end points, maximally serial and maximally parallel versions were generated. These versions established the smallest and largest possible data paths.

(b) To establish a baseline for common case scheduling and allocation, the SM1 was synthesized with uniform instruction frequencies, with and without trace scheduling. To be precise, uniform instruction frequencies mean that, for any transfer of control, the probability of taking any branch is the same (and nonzero).

(c) To further explore common case scheduling and allocation, a set of instruction frequencies was constructed by hand, loosely based on SPEC benchmark data. The SM1 was then synthesized with and without trace scheduling.

(d) To explore the effect of a more severe delay constraint, the SM1 was synthesized with a delay a fifth as large as that used in previous designs.

(e) Finally, for comparison, the SM2, a slightly more complex design, was synthesized using uniform instruction frequencies.

3. Simple Machine Results

The results of synthesis were quantified.

(i) To have an approximate measure of input specification complexity, the number of register transfers produced by the design were counted.

(ii) To have an approximate measure of control (FSM) size, the number of basic blocks and execution cycles generated by the design were counted.

(iii) To measure data path size, the area of the generated data path was computed.

(iv) To measure the overall performance of the design, the cycles per instruction (CPI) performance metric for each design was computed (see Chapter 6).

Synthesis constraints and variables are summarized in Table 10-1. Quantified synthesis results appear in Table 10-2.

Processor	Benchmark	Optimization	Area	Delay	Design Name
SM1	-uniform-	-	-serial-	100	SM1-ser
SM1	-uniform-	-	-parallel-	100	SM1-par
SM1	-uniform-	-	1000000	100	SM1-un
SM1	-uniform-	-trace-	1000000	100	SM1-un-T
SM1	spec	-	1000000	100	SM1-sp
SM1	spec	-trace-	1000000	100	SM1-sp-T
SM1	spec	-	1000000	20	SM1-sp-20
SM2	-uniform-	-	1000000	100	SM2-un

Table 10-1: SM1-Based Designs

Notes:

- Area constraints are in square microns. Delay constraints are in nanoseconds.
- Special constraints include serial (allocate no optional functional units) and parallel (allocate all optional functional units).
- Benchmarks include -uniform- (for any branch, all probabilities are uniform).

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
SM1-ser	20	9	18	398000	4.8
SM1-par	20	9	18	398000	4.8
SM1-un	20	9	18	398000	4.8
SM1-un-T	20	13	19	398000	4.9
SM1-sp	20	9	18	398000	5.4
SM1-sp-T	20	13	19	398000	5.5
SM1-sp-20	20	9	18	453600	5.4
SM2-un	41	30	38	544400	4.3

Table 10-2: SM1 Results

Notes:

- The cycle count is the number of distinct cycles in the design. There are fewer blocks than cycles (some blocks require more than one cycle to execute), and fewer cycles than register transfers (some register transfers execute in parallel).
- Size is the size of the data path, excluding buses, in square microns.

A number of observations follow from these experiments.

(1) The maximally serial and maximally parallel data paths are identical. They have an ALU, a one bit shifter, four registers, including a memory interface, and three buses.

(2) The only difference between the uniform frequency versions is one more block in the trace scheduled design, and hence a larger CPI (in general, more blocks with the same number of register transfers means reduced scheduling freedom and lessened concurrency).

(3) The SPEC benchmark designs, because of weighting, have worse performance (higher CPI) than the uniform frequency versions; they are otherwise identical.

(4) With a 100 nanosecond delay constraint the system selects a ripple carry ALU; with the 20 nanosecond constraint a carry bypass ALU is used, requiring more area.

(5) The SM2 is larger in all ways; its data path has two more registers and a comparator, compared to the SM1. It is nonetheless still a simple machine.

In sum, the SM1 has served as an adequate basic test of the Viper system, but is so simple that little opportunity exists for optimization. There are virtually no differences between the various synthesized designs.

Chapter Eleven: 6502 Results

This chapter introduces the 6502, discusses synthesis experiments using the 6502, and presents the synthesis results.

1. 6502 Specifications

More extensive Viper experiments are based on the 6502 microprocessor. The 6502 was chosen because it is both relatively small and relatively complicated. It has 61 instructions, 11 addressing modes, and four types of interrupts.

The complete specification is given in [Viper]. Highlights are presented here.

The specification uses nine registers and 13 fields. The registers and fields are of varying sizes and widths.

```
stateRegister(acc, 8).
stateRegister(x, 8).
stateRegister(y, 8).
stateRegister(pc, 16).
stateRegister(sp, 8).
stateRegister(p, 8).
stateRegister(memAR, 16).
stateRegister(memDR, 8).
stateRegister(t, 16).
stateField(p, nflag, 7).
stateField(p, vflag, 6).
stateField(p, bflag, 4).
stateField(p, dflag, 3).
stateField(p, iflag, 2).
stateField(p, zflag, 1).
stateField(p, cflag, 0).
stateField(pc, highpc, (15 - 8)).
stateField(pc, lowpc, (7 - 0)).
stateField(memAR, highaddr, (15 - 8)).
stateField(memAR, lowaddr, (7 - 0)).
stateField(t, highbyte, (15 - 8)).
stateField(t, lowbyte, (7 - 0)).
```

Eight of the registers are required; the t register is used internally for temporaries. The single bit fields of the p register are required. Since the memory interface is just eight bits wide, high and low byte fields are needed for the 16 bit registers.

The basic tail recursive fetch-execute cycle is considerably more complicated than that of the SM1.

```
run :-
        interrupt,
        fetchI(Inst, Mode),
        stateUpdate,
        fetchO(Mode),
        stateUpdate,
        incrementPC(Mode),
        execute(Inst),
        stateUpdate,
        run.
run.
```

First, interrupts, if any, are handled. Second, the instruction opcode byte is fetched. Third, the operand is fetched, the mode and length of which depend on the opcode. Fourth, the program counter is incremented, the size of which also depends on the opcode. Finally, the instruction is executed.

As an intermediate test of Viper, a simple subset specification of the 6502 was created, with 11 instructions (lda, sta, adc, and, lsr, jsr, rts, jmp, bmi, nop, and rti), four addressing modes (abs, imm, ind, and zerop), and three types of interrupts (int, reset, and halt). The original, complete 6502 specification was edited, with the extra instructions and addressing modes removed.

Many of the experiments used instruction frequencies based on instruction counts derived from SPEC benchmarks. The first part of the gcc data file appears, for example, as

```
dataSource(gcc).
order(interrupts,[true,int,nmi,reset,halt]).
order(instructions,[lda,adc,inc,ora,and,bne,sta,beq,ldx,cmp,rol,inx,tax,ldy,
        stx,sbc,cpx,jmp,rts,jsr,dec,pla,pha,iny,cpy,tay,sty,bit,ror,lsr,dex,
        bcc,bcs,asl,txa,bmi,bpl,tya,dey,eor,txs,tsx,sei,sed,sec,rti,plp,php,
        nop,clv,cli,cld,clc,bvs,bvc,brk]).
order(addressing,[accum,abs,imm,zerop,absx,indx,zeropx,absy,indy,zeropy,ind]).
count(lda,1423040).
count(adc,968723).
count(inc,697446).
...
```

More information about these frequencies and their derivation can be found in Appendix B and in [Viper].

2. 6502 Experiments

The 6502 and its subset was synthesized a number of times using different constraints and benchmark inputs.

(1) To determine design end points, maximally serial and maximally parallel versions of the subset were generated. These versions established the smallest and largest possible data paths.

(2) To establish a subset baseline for common case scheduling and allocation, the subset was synthesized with uniform instruction frequencies, with and without trace scheduling.

(3) To explore the effect of a more severe delay constraint, the subset was synthesized with a delay a fifth as large as that used in previous designs.

(4) To further explore common case scheduling and allocation, three sets of subset instruction frequencies were constructed by hand, based on SPEC benchmark data. The subset was then synthesized with and without trace scheduling, using these three data sets.

(5) To test utility constraints and design subsets, a complete set of 6502 instruction frequencies was created with the frequencies of the extra (unused, non-subset) instructions set to zero. The original, complete specification was then synthesized with this set and with the optional utility constraint defined to ignore instructions with zero frequencies. The same frequency set and utility constraint were then used in combination with keep declarations to generate a slightly more complete processor than the subset; all interrupts and all addressing modes were kept.

(6) To establish a baseline for common case scheduling and allocation using the complete 6502, the complete specification was synthesized with uniform instruction frequencies, with and without trace scheduling.

(7) To explore common case scheduling and allocation, three sets of 6502 instruction frequencies were constructed, based on SPEC benchmark data. The complete specification was then synthesized with and without trace scheduling, using these three data sets.

(8) To further explore design subsets, the three SPEC data sets were used in conjunction with a utility constraint of one percent to generate three 6502 subsets. In these designs all interrupts, addressing modes, and four important instructions (rti, jsr, rts, and jmp) were kept.

3. 6502 Results

The results of synthesis were quantified using the same metrics as those used for the SM1.

The simple subset constraints and synthesis results appear in Tables 11-1 and 11-2, and the constraints and results for the complete 6502 appear in Tables 11-3 and 11-4.

Processor	Benchmark	Optimization	Area	Delay	Design Name
s6502	-uniform-	-	-serial-	100	S-ser
s6502	-uniform-	-	-parallel-	100	S-par
s6502	-uniform-	-	1000000	100	S-un
s6502	-uniform-	-trace-	1000000	100	S-un-T
s6502	-uniform-	-	1000000	20	S-un-20
s6502	gcc	-	1000000	100	S-cc
s6502	gcc	-trace-	1000000	100	S-cc-T
s6502	eqntott	-	1000000	100	S-eq
s6502	eqntott	-trace-	1000000	100	S-eq-T
s6502	spice	-	1000000	100	S-sp
s6502	spice	-trace-	1000000	100	S-sp-T
6502	-uniform-	-0%-subset-	1000000	100	6502-s
6502	-uniform-	-0%-keep-	1000000	100	6502-k

Table 11-1: Simple Subset 6502 Designs

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
S-ser	122	20	54	480236	10.24
S-par	122	20	54	480236	10.24
S-un	122	20	54	480236	10.24
S-un-T	252	46	107	620636	8.57
S-un-20	122	20	54	535836	10.24
S-cc	122	20	54	480236	11.39
S-cc-T	252	46	107	620636	9.38
S-eq	122	20	54	480236	10.08
S-eq-T	144	36	64	620636	8.62
S-sp	122	20	54	480236	12.55
S-sp-T	252	46	106	480236	9.36
6502-s	122 (340)	20 (79)	64 (171)	480236	10.24
6502-k	164 (340)	27 (79)	78 (171)	539436	10.24

Table 11-2: Simple Subset 6502 Results

Notes:

- The trace scheduled designs involve 40 traces.
- For automatically generated subset designs (using utility constraints), the transfer, cycle, and block numbers include implemented counts and total counts (in parentheses).

Processor	Benchmark	Optimization	Area	Delay	Design Name
6502	-uniform-	-	1000000	100	6502-un
6502	-uniform-	-trace-	1000000	100	6502-un-T
6502	gcc	-	1000000	100	6502-cc
6502	gcc	-trace-	1000000	100	6502-cc-T
6502	eqntott	-	1000000	100	6502-eq
6502	eqntott	-trace-	1000000	100	6502-eq-T
6502	spice	-	1000000	100	6502-sp
6502	spice	-trace-	1000000	100	6502-sp-T
6502	gcc	-1%-keep-	1000000	100	6502-cc-s
6502	eqntott	-1%-keep-	1000000	100	6502-eq-s
6502	spice	-1%-keep-	1000000	100	6502-sp-s

Table 11-3: Complete 6502 Designs

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
6502-un	340	79	171	601744	11.28
6502-un-T	473	161	226	742144	10.29
6502-cc	340	79	171	601744	9.38
6502-cc-T	473	161	226	742144	7.73
6502-eq	340	79	171	601744	9.31
6502-eq-T	473	161	226	601744	7.69
6502-sp	340	79	171	601744	9.37
6502-sp-T	473	161	225	742144	7.53
6502-cc-s	215 (340)	40 (79)	103 (171)	538992	9.18
6502-eq-s	179 (340)	33 (79)	87 (171)	518192	9.21
6502-sp-s	192 (340)	36 (79)	94 (171)	538992	9.23

Table 11-4: Complete 6502 Results

Note: the trace scheduled designs involve 158 traces.

A number of observations can be made from these results.

(1) Moderate concurrency exists in the non trace scheduled designs. On the average, two register transfers are executed per cycle.

(2) This concurrency is not due to multiple functional units, however, but multiple data transfers per cycle (the subset serial and parallel designs have the same data path).

(3) Performance increases with the trace scheduled designs, as well as the size of the design (number of register transfers).

(4) This increase is to some extent due to more concurrency via multiple functional units (two ALUs instead of one), but not entirely.

(5) The trace scheduler reorganizes the designs so that address computations are overlapped with instruction execution (in some sense pipelining them). The exact structure of this reorganization (and the need for multiple ALUs) depends on the relative importance of various instructions and addressing modes.

(6) Different benchmarks cause fairly wide differences in performance measures, with maximum differences of 25% (6502 subset, non trace scheduled), 9% (6502 subset, trace scheduled), 21% (complete 6502, non trace scheduled), and 37% (complete 6502, trace scheduled).

(7) As with the SM1, a 20 nanosecond delay constraint causes a faster (carry bypass) ALU to be chosen.

(8) The subset allocation option correctly created the subset 6502 data path.

(9) The subset keep allocation option correctly created a slightly larger subset design with required additional functionality, adding two needed registers (x and y) to the data path.

(10) The one percent subsets were also correctly created. For gcc, eqntott, and spice, 33, 40, 37 instructions were omitted respectively. Performance was not substantially altered from the complete designs, which is to be expected, since the weight of the omitted instructions is small. Data path area, however, was decreased 10% to 14%, transfer count 37% to 47%, block count 49% to 58%, and cycle count 40% to 49%.

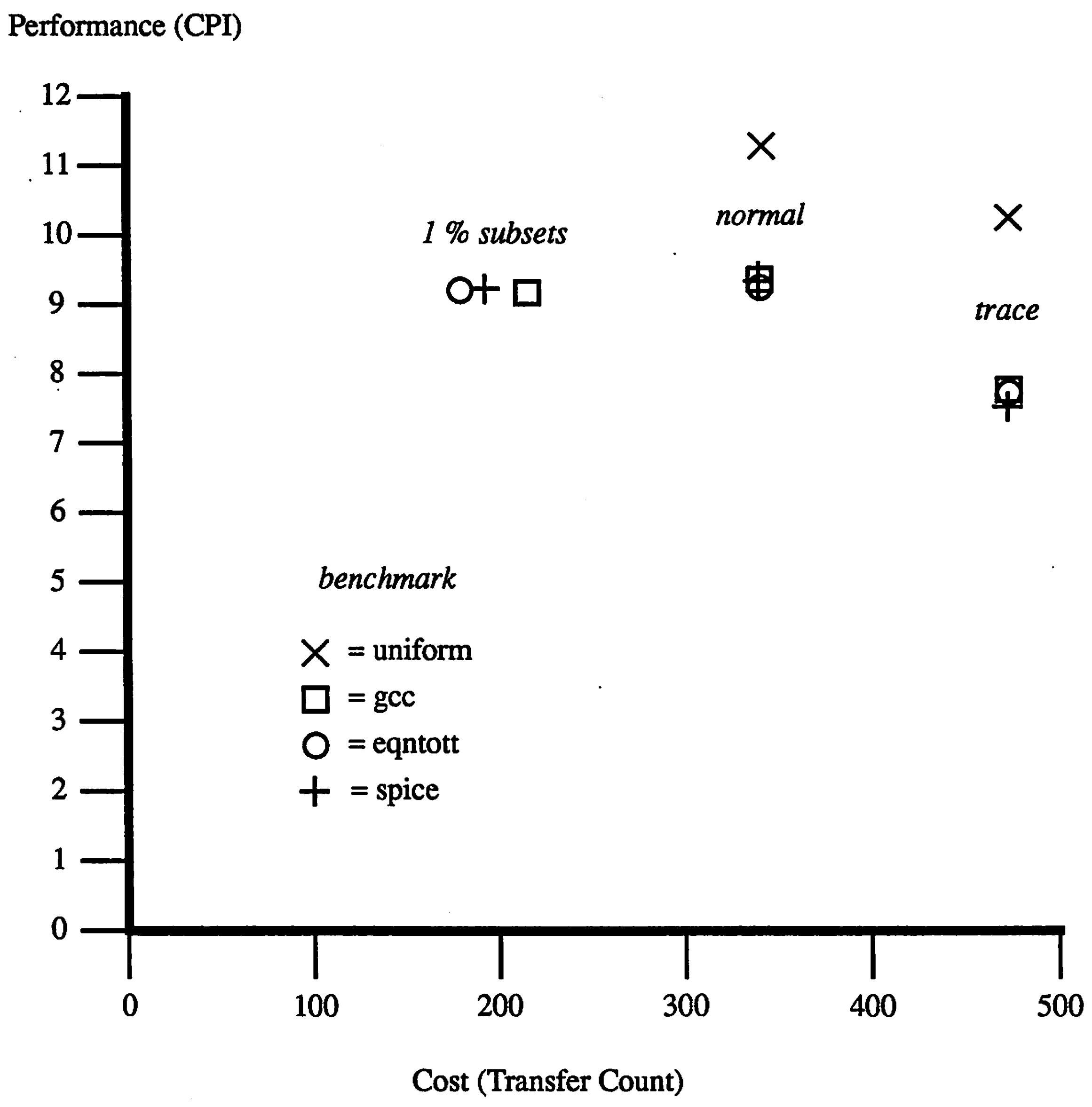


Figure 11-1: 6502 Individual Synthesis Results

Figures 11-1 and 11-2 pictorially display some of these results, relating cost (in terms of register transfers) to performance (CPI). Figure 11-2 plots the geometric mean of all benchmarks for each type of design (subset, normal, and trace scheduled); in terms of cost-performance, normal designs are better than trace scheduled ones. Note that the one percent subset designs are not faster than the normal designs, and are slower than the trace scheduled ones, but are substantially smaller.

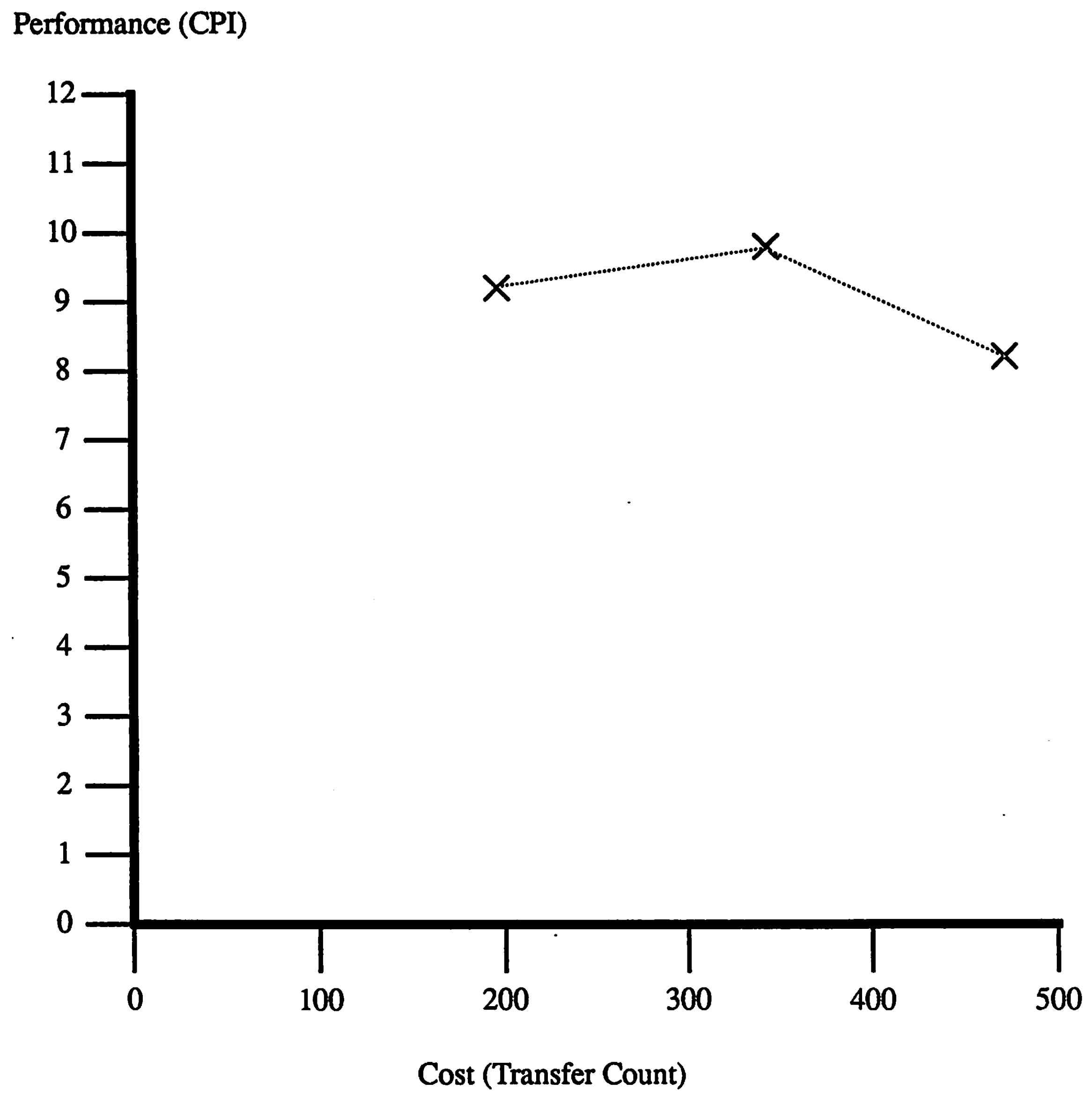


Figure 11-2: 6502 Average Synthesis Results

4. Common Case Scheduling Summary

The most interesting differences in designs are between trace scheduled and non trace scheduled versions. Trace scheduling increases performance, but at some cost in control path size (number of cycles) and data path size. Comparison of these quantities is summarized in Table 11-5. In general, the percentage CPI improvements are less than the increased costs.

Design Name	% Cycle Count Increase	% Size Increase	% CPI Improvement
S-un	98	29	19
S-cc	98	29	21
S-eq	19	29	17
S-sp	96	0	34
6502-un	32	23	10
6502-cc	32	23	21
6502-eq	32	0	21
6502-sp	32	23	24

Table 11-5: A Comparison of Normal and Trace Scheduled 6502 Designs

Chapter Twelve: BAM Results

This chapter describes the Berkeley Abstract Machine (BAM) and its implementation, discusses synthesis experiments using the BAM, and presents synthesis results.

1. BAM Specifications

The BAM is a contemporary general purpose microprocessor minimally extended to support Prolog ([BAM] and [BAM-Manual]). It has register files and just one addressing mode, in contrast to the 6502. The structure of its fetch-execute loop is identical to that of the SM1. Its support for Prolog primarily consists of:

- a double word memory interface and associated double word instructions (ldd, std, stdc, pushd, pushdc),
- tagged data and associated instructions (ldi, sti, stid, cmpi, lea, btgeq, dref), and
- support for Prolog unification (uni, swb, swt).

Because the BAM itself is an architectural experiment, and because it involves advanced features (such as tags, a double word memory interface, and register files), the Viper experiments using BAM include various architectural alternatives and their resulting measurements, as well as common case optimization tests.

The BAM, as implemented for these experiments, has 44 instructions and one register file containing 32 registers. The specification also uses a program counter, a program status word register, and a temporary register.

Complete BAM specifications are given in [Viper]; highlights are discussed presently.

1.1. Implemented Instructions

A few BAM instructions were not implemented for these experiments, either because they required uninteresting but substantial additions to the functional unit library, or because they raised peripheral specification issues.

The functional unit additions would have been to support 28 bit (tagged) ALU operations; all 28 bit instructions were omitted. Note that several functional units were added to support various instructions: additional comparison tests for cmp and cmpi, unbound tag tests for swb, and multiport memory for double word instructions.

The specification issues involve pipeline instruction annulling (btan and btat), interaction between the processor and the cache (ldl and stu), interaction between the processor and the memory system (las), and trapping (trap and rft). In general, these issues relate to specifying the larger system context, an important problem that is beyond the scope of this work.

The remaining, implemented instructions are: ld, ldx, st, stx, ldi, sti, stid, ldd, std, stdc, push, pusht, pop, pushd, pushdc, add, addi, sub, and, andi, or, ori, xor, xori, sll, slli, sra, srai, srl, srli, umin, umax, cmp, cmpi, bt, jmp, jmpr, call, btgeq, lea, dref, uni, swb, and swt.

Two subsets of these implemented instructions were created.

One was developed to establish a base line for measuring the cost of Prolog support. It consists of all non-Prolog and non-Prolog-inspired instructions (ld, ldx, st, stx, push, pop, add, addi, sub, and, andi, or, ori, xor, xori, sll, slli, sra, srai, srl, srli, cmp, bt, jmp, jmpr, and call).

Another, larger subset was generated for testing the trace scheduler, which does not support a double word memory interface. This subset simply omits the double word instructions (ldd, std, stdc, pushd, and pushdc).

1.2. Register Files

As was described in Chapter 4, Viper supports two models of register file access. The first model has the register index presented to the register file in one cycle, and the value read from or written into the file in the *next* cycle. The second model has the index presented and the value read or written in the *same* cycle. Equivalent BAM specifications were written using both models, and most synthesis experiments were run using both.

In addition, an automatically generated variant of the second model was developed. This variant addresses the situation where a register file is unable to read a value and then write a value in the same cycle. In such read-write cases a latch must be inserted between the read and the write.

Consider the add instruction using the second model. It reads two operands from the register file and stores the result, and appears thus:

```
execute(add) :-
    access(memDR, opnd1, I),
    rval(port(r, r1), I, RI),
    access(memDR, opnd2, J),
    rval(port(r, r2), J, RJ),
    Result is RI + RJ,
    access(memDR, opnd3, K),
    wval(port(r, w), K, Result).
```

Assuming that the result cannot be stored in the same cycle, the specification would be rewritten thus:

```
execute(add) :-
    % cycle 1
    access(memDR, opnd0, I),
    rval(port(r, r1), I, RI),
    access(memDR, opnd2_1, J),
    rval(port(r, r2), J, RJ),
    Result is RI + RJ,
    set(elatch, Result),
    % cycle 2
    access(elatch, RK),
    access(memDR, opnd1, K),
    wval(port(r, w), K, RK).
```

Specifications in the first form can be manually translated into the second, but an automatic method, operating at the register transfer level, was developed. Viper has a tool (in the mort optimization module -- see Chapter 9) that scans for sequences of read and writes, and inserts latches as needed (the tool also recomputes exprt lists -- see Chapter 5).

For example, the RTL representation of the add instruction with read and write in the same cycle, is:

```
rtran(25,block(7),field(memDR,opnd0),port(r,r1),rval,tempval1).
rtran(26,block(7),field(memDR,opnd2_1),port(r,r2),rval,tempval2).
rtran(27,block(7),tempval1,tempval2,+,tempval3).
rtran(28,block(7),field(memDR,opnd1),tempval3,wval,port(r,w)).
```

It is transformed into:

```
rtran(25,block(7),field(memDR,opnd0),port(r,r1),rval,tempval1).
rtran(26,block(7),field(memDR,opnd2_1),port(r,r2),rval,tempval2).
rtran(27,block(7),tempval1,tempval2,+,elatch).
rtran(28,block(7),field(memDR,opnd1),elatch,wval,port(r,w)).
```

1.3. Specification Structure

Because the BAM is a relatively simple machine with a simple single word instruction fetch, and in order to explore the effect of specification structure on optimization, versions of the BAM with instruction fetch in various places were constructed.

These were: a) the conventional instruction fetch before instruction execution -- *leading fetch*,

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

b) fetch after execution -- *trailing fetch* (which can benefit the trace scheduling process),

```
run :-
    % this assumes an initial fetch, before run is called
    access(memDR, opcode, OP),
    execute(OP),
    fetch,
    run.
run :- true.
```

c) fetch with each instruction -- *distributed fetch* (which leads to greater concurrency at the cost of duplicate register transfers),

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

and d) a combined approach, using distributed fetch only with commonly executed instructions (the less common instructions using a single trailing fetch) -- *mixed fetch* (which avoids the cost of duplicate register transfers for seldom used instructions).

2. BAM Experiments

Several different versions of the BAM were synthesized. The designs were all synthesized with a delay constraint of 33 nanoseconds (the cycle time of the manually-designed BAM chip, which was fabricated and runs at speed).

(1) To explore the relative cost of Prolog support, BAM versions were synthesized with 1) no Prolog support, 2) Prolog support minus double word instructions, and 3) complete Prolog support.

(2) To investigate the different costs of the different register file models, versions using the first model, the second model, and the variant of the second model, described above, were synthesized.

(3) To determine the possible advantage of full dual ported memory, in contrast to simply a wider bus to memory, versions employing both mechanisms were generated.

(4) To test the efficacy of common case allocation, some designs were synthesized using an area constraint that would limit the allocation of optional functional units.

(5) To explore common case scheduling and allocation and get benchmark-based performance measurements, some designs were synthesized using instruction frequencies derived from the Prolog benchmark composite.

(6) To determine the cost of instruction execution only and assuming overlapped instruction fetch, designs were synthesized from partial specifications in which instruction fetch was omitted entirely.

(7) To test the utility of trace scheduling on the BAM specification, trace scheduled designs were generated.

(8) To explore the effect of specification structure on trace scheduling, leading and trailing fetch versions of the BAM were synthesized using trace scheduling.

(9) To investigate a manual alternative to trace scheduling, and for comparison with the immediately preceding experiment, distributed fetch versions of the BAM were synthesized.

(10) To explore a cost effective refinement of distributed fetch, mixed fetch versions of the BAM were synthesized. A special version of the translator (the tran module -- see Chapter 9) was constructed to perform this optimization.

3. BAM Results

The results of synthesis were quantified using the same metrics as those used for the SM1 and 6502. Since the area and delay constraints were the same for all designs, the tables of results are simpler than their SM1 and 6502 counterparts. In the tables, the name of each design encodes the features it has, using the following abbreviations.

Feature	**Abbreviation**
no Prolog support	nP
single word Prolog support	Ps
double word Prolog support	Pd
two cycle register file access	2c
one cycle access, one per cycle	1c1
one cycle access, two per cycle	1c2
single word memory interface	sw
double word memory interface	dw
double port memory interface	dp
leading instruction fetch	lf
trailing instruction fetch	tf
distributed instruction fetch	df
mixed instruction fetch	mf
no instruction fetch	nf
trace scheduled	ts

Table 12-0: BAM Feature Abbreviations

Note: every design name contains, in order, one abbreviation from the first four categories.

The various architectural variants (register file models, instruction sets, and memory systems) are presented in Tables 12-1 and 12-2 and Figures 12-1 and 12-2. Trace scheduling results are shown in Table 12-3. The assorted instruction fetch variant results appear in Tables 12-4 and 12-5 and Figure 12-3. The number of data path elements allocated for each design is summarized in Table 12-6. All the figures are in the form of area-time graphs, using modified area and time metrics.

Unless otherwise indicated, designs were synthesized with unlimited area and uniform instruction frequencies.

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
nP-2c-sw-lf	106	28	67	2674016	4.32
nP-1c1-sw-lf	102	28	57	2792416	4.12
nP-1c2-sw-lf	102	28	40	2674016	3.76
Ps-2c-sw-lf	178	56	113	2758904	5.24
Ps-1c1-sw-lf	176	56	99	2877304	4.91
Ps-1c2-sw-lf	176	56	79	2758904	4.47
Pd-2c-dw-lf	230	61	129	2995704	5.38
Pd-1c1-dw-lf	221	61	116	3280760	5.02
Pd-1c2-dw-lf	221	61	94	3162360	4.57
Pd-2c-dp-lf	230	61	129	3162360	5.38
Pd-1c2-dp-lf	217	61	88	3554360	4.54

Table 12-1: BAM Architectural Variant Design Results

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
Ps-2c-sw-lf	178	56	113	2758904	5.34
Ps-1c1-sw-lf	176	56	99	2877304	4.85
Ps-1c2-sw-lf	176	56	79	2758904	4.60
Pd-2c-dw-lf	230	61	129	2995704	5.82
Pd-1c1-dw-lf	221	61	112	3280760	5.25
Pd-1c2-dw-lf	221	61	90	3162360	4.95
Pd-2c-dp-lf	230	61	129	3162360	5.82
Pd-1c2-dp-lf	217	61	88	3554360	4.93

Table 12-2: BAM Variants Synthesized Using Benchmark Frequencies

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
Ps-2c-sw-lf	178	56	113	2758904	5.24
Ps-2c-sw-lf-ts	197	101	129	2758904	5.29
Ps-1c1-sw-lf	176	56	99	2877304	4.91
Ps-1c1-sw-lf-ts	178	90	110	2877304	5.11
Ps-1c2-sw-lf	176	56	79	2758904	4.47
Ps-1c2-sw-lf-ts	179	73	90	3150904	4.66

Table 12-3: BAM Trace Scheduled Design Results

Note: the trace scheduled designs involve 107 traces.

Design Name	Transfer Count	Block Count	Cycle Count	Size	CPI
Ps-2c-sw-lf-ts	197	101	129	2758904	5.29
Ps-2c-sw-lf	178	56	113	2758904	5.24
Ps-2c-sw-tf-ts	346	104	183	3150904	4.19
Ps-2c-sw-df	304	64	143	2758904	3.79
Ps-2c-sw-nf	175	56	111	2758904	3.24
Ps-1c2-sw-lf-ts	179	73	90	3150904	4.66
Ps-1c2-sw-lf	176	56	79	2758904	4.47
Ps-1c2-sw-tf-ts	328	80	149	3150904	3.60
Ps-1c2-sw-df	308	64	129	3150904	3.46
Ps-1c2-sw-df*	308	64	129	2758904	3.46
Ps-1c2-sw-nf	175	56	77	2758904	2.47
Pd-2c-dp-lf	230	61	129	3162360	5.38
Pd-2c-dp-df	392	69	167	3162360	3.82
Pd-2c-dp-nf	227	61	127	3162360	3.38
Pd-1c2-dp-lf	217	61	88	3554360	4.54
Pd-1c2-dp-df	379	69	144	3554360	3.35
Pd-1c2-dp-nf	214	61	86	3554360	2.54

Table 12-4: BAM Modified Fetch Design Results

Note: the design marked with an asterisk (*) was synthesized with an area constraint of 3000000.

Design Name	Cutoff	Transfer Count	Block Count	Cycle Count	Size	CPI
Pd-2c-dp-tf	100%	230	61	129	3162360	5.82
Pd-2c-dp-mf	2%	290	68	154	3162360	4.79
Pd-2c-dp-mf	0.2%	338	70	166	3162360	4.57
Pd-2c-dp-mf	0%	395	70	169	3162360	4.55
Pd-2c-dp-df	0%	395	70	169	3162360	4.55
Pd-1c2-dp-tf	100%	217	61	88	3554360	4.93
Pd-1c2-dp-mf	2%	277	68	116	3554360	4.13
Pd-1c2-dp-mf	0.2%	325	70	130	3554360	3.94
Pd-1c2-dp-mf	0%	382	70	146	3554360	3.93
Pd-1c2-dp-df	0%	382	70	146	3554360	3.93

Table 12-5: BAM Mixed Fetch Design Results

Note: in the above table, the cutoff field indicates the point at which an instruction was implemented with distributed as opposed to trailing fetch. Specifically, the cutoff value is the execution percentage (in the benchmark composite) that an instruction must have to be implemented with distributed fetch. Thus, at the endpoints, are complete trailing fetch and complete distributed fetch designs. In between are, for example, 2% designs, where every instruction that is executed more than 2% of the time is implemented using distributed fetch. Note that 0% mixed fetch designs are the same as normal distributed fetch designs (with a 0% cutoff every instruction uses distributed fetch).

Data Path Size	ALUs	Incrementers	Memory Interfaces	Latches	Tag Logic
2674016	1	0	1	0	0
2758904	1	0	1	0	1
2792416	1	0	1	1	0
2877304	1	0	1	1	1
2995704	1	0	2	0	1
3150904	2	0	1	0	1
3162360	1	1	2	0	1
3280760	1	1	2	1	1
3554360	2	1	2	0	1

Table 12-6: BAM Data Path Element Allocation

Note: all data paths contain a PC register, a PSW register, a temporary register, a comparator, a shifter, and a register file; the ALUs and counters are carry bypass. (The double word memory interface could be simpler and smaller; the second ALUs could be incrementers.)

A number of observations can be made from these results.

(1) The different register file models yield expected differences in performance, with the more powerful, faster hardware producing faster designs. One cycle (1c2) designs are 17% faster than similar variant (1c1) designs, which are in turn 6% faster than two cycle (2c) designs (from Tables 12-1 and 12-2). One cycle designs have a mean of 2.38 register transfers per cycle, one cycle variants have 1.83 transfers per cycle, and two cycle designs have 1.65 transfers per cycle.

(2) An obvious, but important resulting observation, is that underlying hardware has a significant influence on performance, which may be at least as significant as optimization strategies.

(3) Prolog support has a cost. Designs with single word Prolog support were 20% slower and 3% larger (in data path area) than designs with no Prolog support, and double word designs were 2% slower and 13% larger than single word designs (from Tables 12-1 and 12-2).

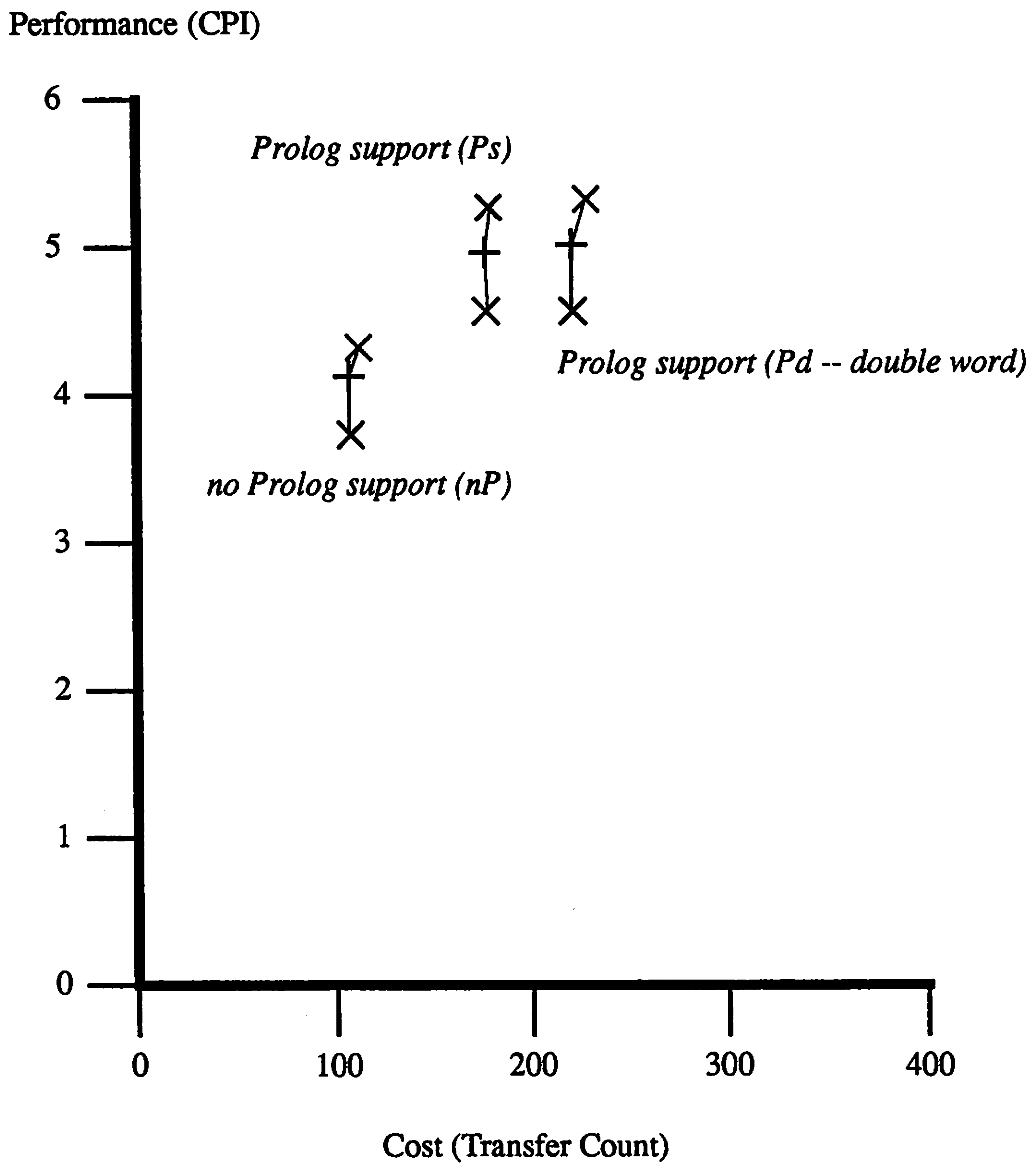
(4) The CPI measure used in the above speed comparisons is to some extent misleading as applied. It correctly shows how much more slowly an average instruction executes, but it does not indicate relative execution times of complete benchmarks (designs with more complex instructions may very well be faster because fewer instructions overall will be executed -- see, for example, the formula in [MIPS], page 1-2). The CPI metric is appropriate for comparing different implementations of the same instruction set, but it is inadequate for comparing designs with different instruction sets.

(5) Full double ported memory produced no performance improvement.

(6) An area constraint, combined with common case allocation, produced a more efficient design. The bounded area distributed fetch design in Table 12-4 (Ps-1c2-sw-df) uses one ALU instead of two, compared to its unbounded alternate. This second ALU is used in the unbounded version for PC increment (in the add, addi, sub, and, andi, or, ori, xor, xori, and lea instructions). In the bounded design, the PC increment is delayed a cycle and occurs in parallel with the instruction memory read. Both the bounded and unbounded designs have the same cycle count and CPI values.

(7) Use of the benchmark composite instruction frequencies, in contrast to uniform frequencies, did not affect the quality of the resulting designs. They did, however, result in higher calculated CPI values, because the commonly used, more heavily weighted instructions, for memory access and Prolog support, are longer instructions (taking more cycles).

(8) Removing the fetch clause to measure only cycles per instruction execution yielded the obvious and anticipated two cycle improvement (instruction fetch requires two cycles, and is always executed). It would be reasonable to pursue overlapped fetch and execution.

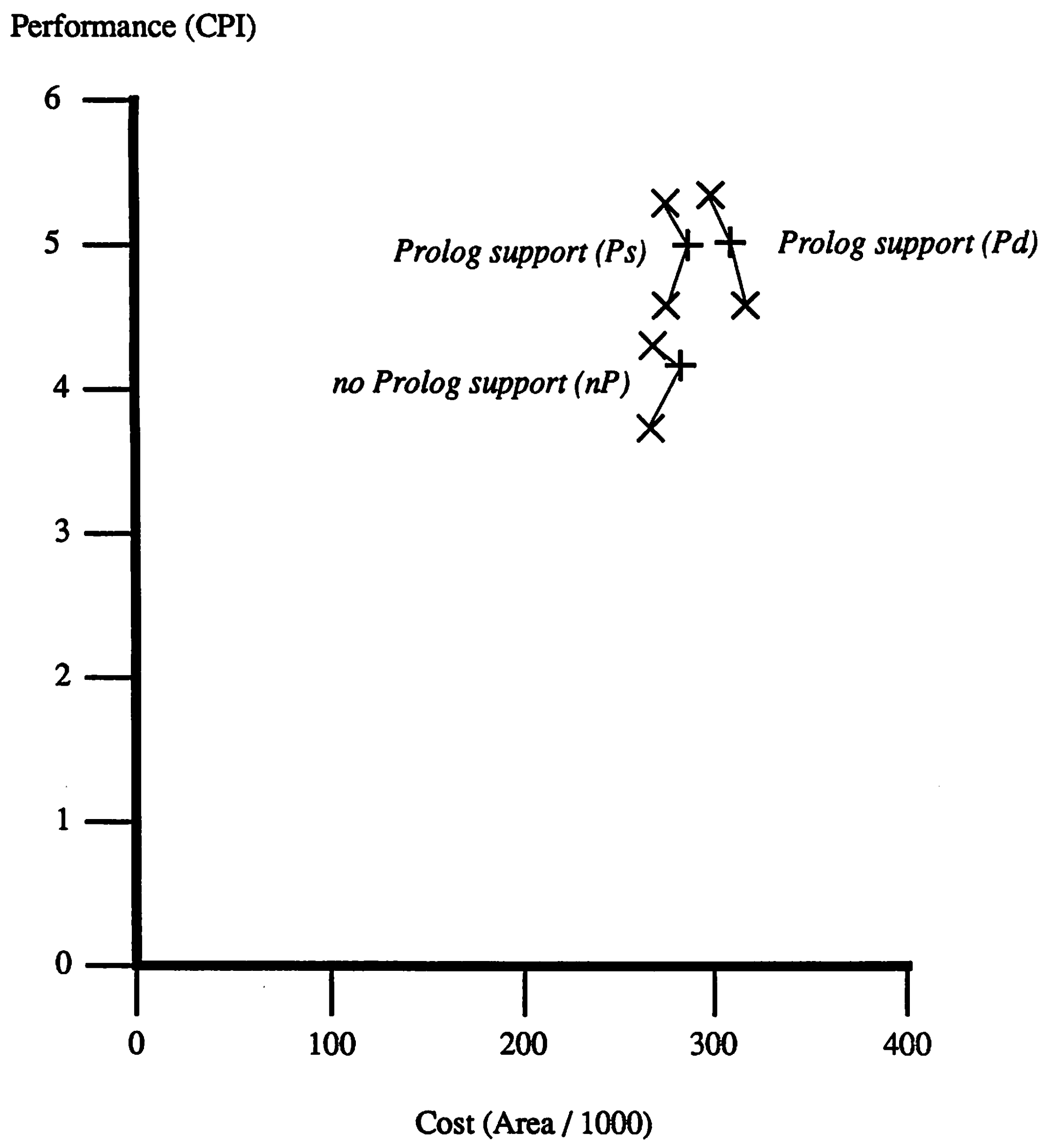


(Each connected set of points includes 2c, 1c1, and 1c2 designs.)

Figure 12-1: BAM Variants (Transfer Count Cost)

(9) Trace scheduling with leading fetch produces results similar to those for the SM1 (since instruction fetch is simple and cannot be effectively merged with instruction execution, there is no improvement).

(10) Trace scheduling with trailing fetch produces substantial improvements in performance, in return for increased cost, in the form of duplicate register transfers; the instruction fetch transfers are replicated in each instruction execution clause. For single word Prolog, trace scheduling causes a 25% CPI improvement and a 94% increase in transfer count for two cycle register files, and a 24% CPI improvement and an 86% transfer count increase for one cycle register files (from Table 12-4).



(Each connected set of points includes 2c, 1c1, and 1c2 designs.)

Figure 12-2: BAM Variants (Area Cost)

(11) Considering the difference in trace scheduling results between leading and trailing fetch, it is apparent that high level specification structure is important, and high level transforms are useful.

(12) The trailing fetch trace scheduling results can be duplicated manually with distributed fetch. For single word Prolog, distributed causes a 38% CPI improvement and a 71% increase in transfer count for two cycle register files, and a 29% CPI improvement and a 75% transfer count increase for one cycle register files (from Table 12-4).

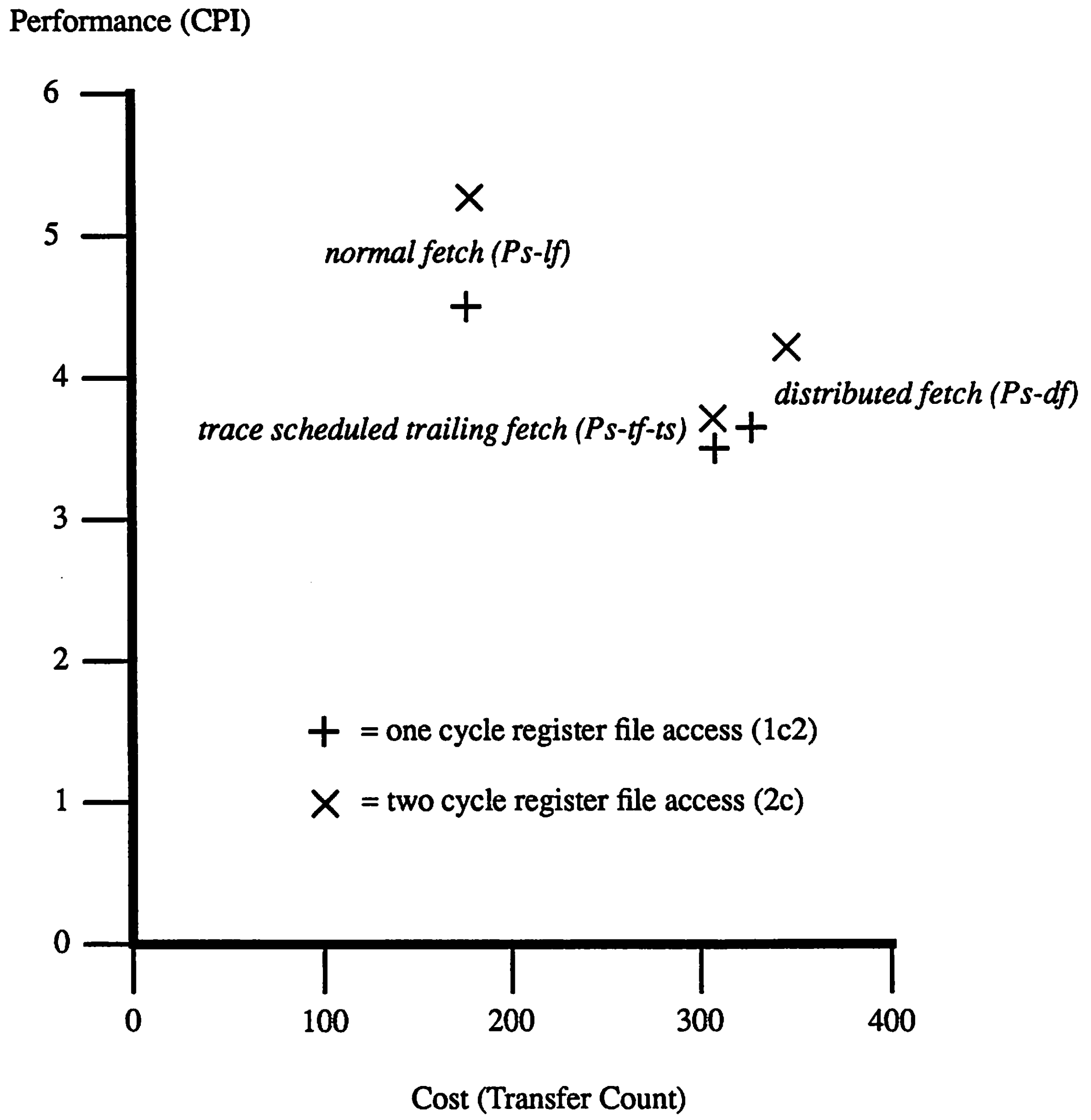


Figure 12-3: BAM Instruction Fetch Variants

(13) The mixed fetch structure produced interesting tradeoffs. At the 2% cutoff 14 instructions were distributed (addi, ld, dref, ldd, st, jmp, cmp, bt, swt, jmpr, btgeq, pushdc, lea, xori); at the 0.2% cutoff 13 more were (umax, ldi, push, call, pusht, pushd, cmpi, ldx, swb, pop, add, std, sti). The following table summarizes the cost-performance results; benefits are in terms of decreased CPI, and costs are in terms of increased transfer count (with increased cycle count for comparison in parentheses). For the two cycle designs, 86% of the benefit comes at 36% (61%) of the cost, and virtually 100% at 65% (94%). For the one cycle designs, 80% of the benefit comes at 36% (48%) of the cost, and virtually 100% at 64% (73%). Full distributed fetch and, by implication, trace scheduling, are not, comparatively, cost effective.

Design	2% Benefit	2% Cost	0.2% Benefit	0.2% Cost	0% Benefit	0% Cost
2c	19%	26% (19%)	22%	47% (29%)	22%	72% (31%)
1c2	16%	27% (32%)	20%	49% (48%)	20%	76% (66%)

Table 12-7: BAM Cost-Performance Data For Mixed Fetch

(14) A comparison with the fabricated BAM chip is difficult, because it is pipelined, and contains additional system functionality. The basic data path is the same; extra incrementers and ALUs in the results presented here are reflected in the fabricated BAM in its instruction pipeline and memory system.

Chapter Thirteen: Pipelining

This chapter describes the extensions made to the Viper system to support pipelining. These extensions are not part of the experimental sequence presented in the bulk of this work (which is concerned with non-pipelined designs), but they are germane because they demonstrate the utility and flexibility of the overall Viper framework, and because one specific extension (see RUP below) uses instruction frequencies.

1. Pipelining Extensions

Pipelining, in Viper terms, means that different instructions execute at the same time, each in a different Viper scheduling cycle (each instruction is defined by a basic block, or set of blocks, of register transfers). Each Viper cycle is equivalent to a pipeline stage. A simple add instruction, for example (including specification, register transfers, and schedule, and using dual ported memory),

```
execute(add) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    mem_read(2),
    access(memDR(2), T), access(ac, AC), A is T+AC, set(ac, A).
rtran(4,block(4),field(memDR(1),address),none,move,memAR(2)).
rtran(5,block(4),memAR(2),none,mem_read(2),memDR(2)).
rtran(6,block(4),memDR(2),ac,+,ac).
cycle(4,block(4),1).
cycle(5,block(4),2).
cycle(6,block(4),3).
```

requires three pipeline stages, one for each register transfer. As an add instruction executes sequentially cycles 1, 2, and 3, other instructions are executing other cycles.

Pipelining affects a design in three ways.

- All instructions must take the same number of cycles to execute. The behavior of each cycle for each instruction must be defined. (Viper does not generate the synchronization hardware needed to support pipeline stalls or forwarding.)
- All instructions should use the same resources in the same (relative) cycles; otherwise, duplicate resources may be allocated. The add instruction above, for example, uses an ALU in cycle 3 (performs a +). Other instructions performing ALU operations should do so in cycle 3 as well; otherwise another ALU will be needed.
- All data references must be synchronized, based on data dependencies. For example, the add instruction reads and writes the accumulator in cycle 3; other instructions using the accumulator (such as conditional branches based on the contents of the accumulator) must take that into account.

The process of normalizing and coordinating instructions required by the above conditions can be complex. Viper does not perform this process. It does, however, support the human designer in this effort, in three ways.

- Viper shows the designer how resources are being used and how values are being referenced, with an additional tool.
- Viper provides an additional operator that allows the human designer to add null cycles to instructions, which aids in synchronizing resource and value use.
- Viper assigns each register transfer to a pipeline stage, based on its cycle assignment.

Additionally, the Viper allocator has been modified to generate hardware for pipelined data paths. It allocates this hardware stage by stage (cycle by cycle), and not block by basic block, as is done for non-pipelined designs. The resources allocated to a cycle must fill the needs of all instructions executing that cycle. Furthermore, unlike non-pipelined designs, operations cannot be postponed a cycle. Thus, since all the operations assigned to a cycle must be supported, and none can be delayed, no operations are optional, and needs analysis is unnecessary.

These four Viper extensions are discussed in more detail below. They required no changes to the basic Viper framework or to data structures. Register transfers, data dependency and scheduling information, functional unit declarations, and instruction frequency information are generated and structured as before. Referring to the Viper modules of Chapter 9, only the alloc module, described in Chapter 8, needed modification. Simulation of pipelined designs, however, would have required substantial changes to the Prolog and RTL interpreters, and is not supported.

1.1. The *delay* Operator

The *delay* operator allows the user to create null cycles explicitly in an instruction definition, in order to normalize the various instructions in a design. The operator absorbs a cycle, and everything appearing after it (in the same basic block) is scheduled after it.

For example, consider the above three cycle add instruction, and a one cycle shift instruction.

```
execute(add) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    mem_read(2),
    access(memDR(2), T), access(ac, AC), A is T+AC, set(ac, A).
execute(shr) :- !,
    access(ac, AC), A is AC>>1, set(ac, A).
```

In order for the shift to use the same pipeline as the add instruction, it must take three cycles, and change the accumulator in the third cycle. Two delays accomplish this.

```
execute(add) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    mem_read(2),
    access(memDR(2), T), access(ac, AC), A is T+AC, set(ac, A).
execute(shr) :- !,
    delay,
    delay,
    access(ac, AC), A is AC>>1, set(ac, A).
```

1.2. Assigning Register Transfers to Pipeline Stages

The process of assigning each register transfer to a pipeline stage starts with each transfer's cycle assignment (as discussed in Chapter 8). Consider again the above add example and its cycle assignments

```
rtran(4,block(4),field(memDR(1),address),none,move,memAR(2)).
rtran(5,block(4),memAR(2),none,mem_read(2),memDR(2)).
rtran(6,block(4),memDR(2),ac,+,ac).
cycle(4,block(4),1).
cycle(5,block(4),2).
cycle(6,block(4),3).
```

These cycle assignments bind each register transfer to a relative cycle within the add instruction's basic block (block 4).

The difference between one of these cycles and a pipeline stage is essentially that a pipeline stage is part of a complete design, whereas a cycle is local to a basic block. In this case, three separate pipeline stages (cycles) will be needed to implement the above three transfers, but they are not the first three stages of the design. The add block is in fact preceded by a block containing a one cycle instruction fetch,

```
rtran(1,block(1),memAR(1),none,mem_read(1),memDR(1)).
rtran(2,block(1),memAR(1),none,inc,memAR(1)).
cycle(1,block(1),1).
cycle(2,block(1),1).
```

The fetch is the first stage of the pipeline, and the above three add cycles correspond to stages two, three, and four.

Thus a transfer's pipeline stage is its cycle assignment incremented by the number of cycles in the blocks that precede it. The stage assignments for the above transfers are

```
stage(1,block(1),1).
stage(2,block(1),1).
...
stage(4,block(4),2).
stage(5,block(4),3).
stage(6,block(4),4).
```

Similarly, the (normalized) shift instruction above is also preceded by the same instruction fetch.

```
rtran(10,block(6),delay,delay,delay,delay).
rtran(11,block(6),delay,delay,delay,delay).
rtran(12,block(6),ac,none,shr(1),ac).
cycle(10,block(6),1).
cycle(11,block(6),2).
cycle(12,block(6),3).
stage(10,block(6),2).
stage(11,block(6),3).
stage(12,block(6),4).
```

The first step of the allocation and binding process described in Chapter 8 has been augmented to perform stage assignment. It first computes an as-soon-as-possible schedule (as described in Chapter 8, with one difference discussed below), which results in cycle assignments. It then examines each block in turn, producing stage assignments for that block's register transfers based on cycle assignments and the stage assignments of preceding blocks. It uses block enabling (*blen*) information, described in Chapter 5, to determine the predecessors of each block.

The ASAP scheduling process for pipelining differs from non-pipelined scheduling in the way certain control operations are handled. In general, a pipelined design does not have a single controlling finite state machine, but rather a separate controller for each pipeline stage. As an instruction migrates through the pipeline, it enables various actions by the different controllers. What this means for scheduling specifically is that certain control operations are free, taking no cycles because they are performed as a matter of course by the stage controllers. In particular, opcode dispatch does not require a distinct cycle, because it is built in to each stage controller. The Viper ASAP scheduler has been modified accordingly.

1.3. Allocating Pipelined Data Paths

The Viper allocator, described in Chapter 8 in terms of seven steps, has been modified to generate pipelined data paths. The seven steps have been changed as follows:

[1] This step performs ASAP scheduling, and is discussed in the previous section.

[2], [3], [4]
These steps are concerned with needs analysis and have been removed.

[5] This step creates functional units. It no longer uses needs analysis, but instead allocates data path elements stage by stage. For each pipeline stage in turn, and instruction by instruction (using blen information), it examines the register transfers assigned to that stage, and creates the functional units needed by those register transfers. It creates them using the smallest first strategy discussed in Chapter 8.

[6] This step binds operators to functional units. It is different in that it no longer creates optional functional units (because none are optional), and no longer modifies cycle assignments (because instructions have been normalized).

[7] This step is unchanged.

1.4. Resource Usage Patterns

Viper provides an additional tool (called RUP) to analyze the resource usage patterns of instructions. It classifies instructions by the resources they use each cycle (functional units, registers, and memory), and weights the resulting usage patterns with instruction frequencies. Each pattern can be thought of as a resource-based reservation table ([Arch-Kogge]). The RUP tool is the only part of Viper that, in generating pipelines, uses instruction frequencies.

Viper simply calculates RUP data; it does not use it. The human user, however, can use the data to normalize instructions, conforming all instructions to the same basic pattern, and basing this pattern on the more heavily weighted instructions.

For each resource usage pattern, the following information is generated:

```
[<opcode-list>] <pattern-index>
        <frequency> <pattern-count> <cycle-length> {<enabling-RT>}
```

where

- *<opcode-list>* is the list of instructions that follow this pattern;
- *<frequency>* is the frequency (probability) of this pattern being used (being the sum of the frequencies of the listed instructions);
- *<cycle-length>* is the total number of cycles in the pattern; and
- *<enabling-RT>* is the index of the register transfer that enables this pattern (see the discussion of blen in Chapter 5).

The other items, *<pattern-index>* and *<pattern-count>*, are essentially bookkeeping information (a unique integer identifying the pattern, and the count of instructions that use it).

For each cycle in each pattern, the following information is displayed:

```
<cycle>/<stage> -<resource>
        R:[<read-list>]
        W:[<write-list>]
```

where

- *<cycle>* is the local cycle number (from stage binding above);
- *<stage>* is the pipeline stage number (from stage binding above);
- *<resource>* is the resource (functional unit or memory) used (if any);
- *<read-list>* is the list of registers read; and
- *<write-list>* is the list of register written.

All but *<cycle>* and *<stage>* may be absent, and there may be multiple resources.

For example, for the above add and shift instructions, the patterns are

```
[add,and] <2>
    0.273973 <2> (3) {rt(4)}
        1|2
            R:[field(memDR(1),address)]
            W:[memAR(2)]
        2|3 -mem(2)
        3|4 -ALU
            R:[ac,memDR(2)]
            W:[ac]
[shr] <3>
    0.0410959 <1> (3) {rt(4)}
        1|2
        2|3
        3|4 -SHF
        R:[ac]
        W:[ac]
```

Note that the pattern used by add is also used by and. Also note that the shift pattern is almost a subset of the add pattern. RUP does not try to merge the two.

RUP analyses of various versions of the BAM can be found in [Viper].

Another useful guide for pipelining (not generated by RUP) is the overall performance metric introduced in Chapter 6 and used extensively in the result chapters. This weighted average instruction cycle length is an important design parameter, along with branch instruction frequency. It serves as an upper bound for pipeline length.

2. A Simple Pipelined Machine

As a test of the above mechanisms, a pipelined version of the SM1 of Chapter 10 was synthesized. The pipelined version differs in having normalized instructions with delay operators, and in having a no-op instruction for delayed branches (the other instructions, carried over from the SM1, are add, and, shr, load, stor, jmp, brn, and halt). The complete specification can be found in Appendix A.

The four stages of the resulting pipeline are:

(1) instruction fetch and PC increment;

(2) operand decode and PC modification for branch instructions;

(3) memory read or write; and

(4) ALU operation and accumulator read and write.

One delay slot is needed for branch instructions -- fetch occurs in stage 1, and the PC is modified in stage 2. In addition, two preceding no-ops may be needed for the stor and brn instructions, in order for them to read the correct accumulator values; they reference the accumulator in stage 2, and it is modified in stage 4.

The resulting data path has

- two ALUs, one for PC increment and one for instruction execution (the first could be an incrementer);
- two sets of memory address and data registers, one for instruction fetch and one for instruction execution;
- a shifter; and
- an accumulator and program counter.

The CPI performance of this pipelined SM1, using the instruction frequencies of Chapter 10, is 1.79 (including the contribution of the extra cycles needed by the store and branch instructions), making it three times faster than the non-pipelined version. The data path area is 786016 square microns (using a 100 nanosecond delay constraint), almost twice the size of (388016 square microns more than) the non-pipelined version. The net performance/cost improvement is 1.5.

3. An Alternate Pipelining Technique

The technique described above, using the non-pipelined version of Viper as a starting point, begins with a schedule (heavily influenced by the human designer via the delay operator), and generates a data path.

An alternate technique would be to start with a data path and schedule operations on it. Such a technique would tentatively involve four steps, some of which could be performed by human designers.

(a) Generate resource usage patterns for non-pipelined versions, to get preliminary information about the design.

(b) Build a pipelined data path (perhaps manually), using resource usage information, branch frequency, and CPI metrics.

(c) Schedule register transfers on the data path, perhaps decomposing register transfers. Use the concept of mobility and bypass stages to fit short sequences. The weight (and extent by cycles) of non-fitting instructions is the figure of merit (the smaller the better).

(d) Improve the design (perhaps manually). Add hardware or use internal opcodes to fit long sequences. Iterate to the previous step.

4. Other Pipelining Issues

One potential difficulty with the Viper approach is that Viper register transfers are not always the correct atomic units for pipeline scheduling. Most current pipelined microprocessors further divide such transfers into additional cycles.

For example, the pipelined BAM microprocessor ([BAM]) has a five stage pipeline: instruction fetch, register read, ALU operation, memory operation, and register write. Some of these actions, like instruction fetch, are complete procedures in input specifications that take full cycles to execute. Others, like register read and write, are parts of cycles. The problem is that each pipeline stage should take roughly the same amount of time. Commercial microprocessors address this issue, balancing the time taken by each stage. Any sophisticated pipeline synthesizer must do this too, using register access time, memory delay, and ALU speeds, as well as the technique of register result forwarding. A synthesizer should be able, for example, to decide between a four stage I-R-A-W and five stage I-R-A-M-W pipeline, determining the value of the extra memory stage. It should do this using implementation cost and performance metrics.

Another issue arises with specification. There traditionally was a clear distinction between instruction set architectures (ISAs) and microarchitectures (MAs). ISAs are implementation independent processor specifications, used by system programmers and compiler writers. MAs are processor implementations by and for hardware designers. This distinction exists for the DEC VAX and IBM 370 architecture, for example, which are conventional ISAs. Microprocessors have complicated the situation, however, particularly those with the RISC label, because they deliberately expose implementation details, such as pipeline latency, to the compiler writer. Thus the starting point of Viper synthesis, the ISA, is inherently a bit hazy.

Implementation-independent specifications -- specifications that imply no implementation -- can no longer be expected, at least for the domain of current microprocessors. Synthesizers must be able to optimize specifications with explicit implementation details, particularly exposed pipelines and pipeline delay slots. The BAM, for example, has five instructions (btgeq, swb, swt, btan, and btat) that require instruction annulling.

Other variables also affect pipelining, and should be taken into account by synthesizers. The number of register ports, saving and restoring the pipeline (for interrupts), dealing with register file data dependencies, and the frequency of branch instructions all affect pipeline architecture, and are hard to handle automatically.

Chapter Fourteen: Conclusions and Future Work

1. Summary of Results

To review, the Viper high-level hardware synthesis system has been constructed, which produces data and control paths from behavioral specifications written in a subset of Prolog. Execution counts derived from benchmark programs are used to guide optimization during synthesis. The system uses trace scheduling to optimize designs, and employs an optimizing data path allocation technique that allocates optional functional units in order of importance (that is, frequency of use).

Various non-pipelined versions of three microprocessors have been synthesized: the SM1 (a simple test case), the 6502, and the BAM (a contemporary general purpose microprocessor minimally extended to support Prolog). Synthesis variables included: different execution counts from different benchmarks (to see how benchmarks affected final designs); various cost (area) and performance (delay) constraints; the use, or not, of trace scheduling (to determine its effect on performance and cost); and changes to the BAM specification (to investigate the effects of specification structure on synthesis and optimization).

Resulting synthesized designs were evaluated and compared in terms of data path area, size of control, and cycles per instruction (CPI). The results were not contrasted with results from other systems, because the specifications and functional units used in this work are unique.

The primary 6502 synthesis results were:

(1-1) Moderate concurrency exists in the non trace scheduled designs. On the average, two register transfers are executed per cycle.

(1-2) This concurrency is not due to multiple functional units, but to multiple data transfers per cycle (the subset serial and parallel designs have the same data path).

(1-3) The trace scheduler reorganizes the designs so that address computations are overlapped with instruction execution (in some sense pipelining them). The exact structure of this reorganization (and the need for multiple ALUs) depends on the relative importance of various instructions and addressing modes.

(1-4) In general, the percentage CPI improvements with trace scheduling are less than the increased costs. Also, different benchmarks cause fairly wide differences in CPI measures, with maximum differences of 37%.

(1-5) The data path allocator can automatically create a reduced data path, ignoring instructions that are utilized below some threshold amount (a utility constraint). Some instructions, however, may be important although seldom used (such as return from interrupt), and may be explicitly retained (with keep declarations). To test these features, a set of 6502 instruction frequencies was created with unused instructions (with counts of zero). The complete specification was then synthesized with this set and with the utility constraint defined to ignore instructions with zero counts. The same frequency set and utility constraint were then used in combination with keep declarations to generate a slightly more complete processor than the subset; all interrupts and all addressing modes were kept. These designs were correctly generated.

(1-6) To further explore reduced designs, three benchmark data sets were used in conjunction with a utility constraint of one percent to generate three 6502 design subsets. In these designs all interrupts, addressing modes, and four important instructions (rti, jsr, rts, and jmp) were kept. The one percent subsets were correctly created. For gcc, eqntott, and spice, 33, 40, 37 instructions were omitted respectively. Performance was not substantially altered from the complete designs, which is to be expected, since the weight of the omitted instructions is small. Data path area, however, was decreased 10% to 14%, and control 37% to 47%.

Benchmark Name	% Cycle Count Increase	% Data Path Size Increase	% CPI Improvement
uniform	32	23	10
cc	32	23	21
eqntott	32	0	21
spice	32	23	24

Table 14-1: 6502 Result Summary

The primary BAM synthesis results were:

(2-1) Prolog support has a cost. Designs with Prolog support, using a single word memory interface, were 20% slower and 3% larger (in data path size) than designs with no Prolog support, and designs with a double word interface were 2% slower and 13% larger than single word designs.

(2-2) The speed comparisons of (2-1), based on CPI, are to some extent misleading as applied. They correctly show how much more slowly an average instruction executes, but they do not indicate relative execution times of complete benchmarks (designs with more complex instructions may very well be faster because fewer instructions overall will be executed -- see, for example, the formula in [MIPS], page 1-2). The CPI metric is appropriate for comparing different implementations of the same instruction set, but it is inadequate for comparing designs with different instruction sets.

(2-3) The BAM microprocessor was synthesized with different register files, some requiring two cycles per access, some one cycle. The different register file models yield expected differences in performance, with the more powerful, faster hardware producing faster designs. Fast one cycle designs were 17% faster than slower one cycle variant designs, which were in turn 6% faster than two cycle designs. One cycle designs have a mean of 2.38 register transfers per cycle, one cycle variants have 1.83 transfers per cycle, and two cycle designs have 1.65 transfers per cycle.

(2-4) Trace scheduling produces substantial improvements in performance, in return for increased cost, in the form of duplicate register transfers; the instruction fetch transfers are replicated in each instruction execution clause. For single word Prolog, trace scheduling causes a 25% CPI improvement and a 94% increase in transfer count for two cycle register files, and a 24% CPI improvement and an 86% transfer count increase for one cycle register files.

(2-5) In an attempt to reduce the cost of optimization, an alternative to trace scheduling was developed, which replicates instruction fetch only in commonly executed instructions (defined by a user supplied cutoff threshold); instruction fetch is thus overlapped for common instructions (for speed), and not overlapped (duplicated) for uncommon ones, which saves control size. At a 2% cutoff 14 instructions had overlapped fetch; at the 0.2% cutoff 13 more had it. The following table summarizes the cost-performance results; benefits are in terms of decreased CPI, and costs are in terms of increased transfer count (with increased cycle count for comparison in parentheses). For the two cycle designs, 86% of the benefit comes at 36% (61%) of the cost, and virtually 100% at 65% (94%). For the one cycle designs, 80% of the benefit comes at 36% (48%) of the cost, and virtually 100% at 64% (73%).

Design	2% Benefit	2% Cost	0.2% Benefit	0.2% Cost	0% Benefit	0% Cost
two cycle	19%	26% (19%)	22%	47% (29%)	22%	72% (31%)
one cycle	16%	27% (32%)	20%	49% (48%)	20%	76% (66%)

Table 14-2: BAM Result Summary

Viper was also extended to generate simple pipelined designs; a pipelined version of the SM1 was generated, which was three times faster than the non-pipelined version, with a net performance/cost improvement of 1.5.

2. Conclusions

The basic question asked in Chapter 1 was, *how effectively can the quality of automatically generated microprocessors be improved through the use of instruction frequency statistics?*

In the course of this work, this question has become, *how effective are common case (trace) scheduling and allocation?* The simple, qualitative answer is, *common case scheduling has noticeably increased performance, but common case allocation has had a marginal effect.*

Common case scheduling increased the performance of the synthesized hardware by increasing concurrency (primarily by moving instruction fetch and address computations), at a cost of proportionally greater control size and data path area. The resulting hardware performance was better than any obtainable using normal, intra-block scheduling techniques. Common case scheduling is more effective with complex designs (which take advantage of common case scheduling's bookkeeping). The technique also has the advantage of being general, automatic, and relatively efficient. It is not tuned to certain high-level constructs (such as loops), and does not require human guidance.

Common case allocation demonstrated little applicability in microprocessor synthesis. Opportunities for multiple functional unit concurrency were rare, and virtually all functional units were required. The allocator did, however, make possible the easy, automatic generation of subset microprocessors, tailored to particular applications.

The primary contributions of this work are:

- the successful application of trace scheduling to high-level synthesis, for general, automatic, usage-driven inter-block performance improvements; and
- the development of common case allocation, which can be used with any block-by-block allocation scheme, and which demonstrated its utility in generating microprocessor subsets.

3. Observations

Microprocessor synthesis is a reasonable synthesis domain in which to explore common case optimizations, because microprocessor specifications are rich in unequally weighted alternatives -- different instructions and addressing modes. If there were fewer alternatives, or if the alternatives were usually weighted equally, design space explorable by common case optimization would be considerably smaller.

Also, designs must be relatively complex (and have the proper form) before common case scheduling transformations show any benefit. The SM1 was not improved, but the 6502 was. The mean speed (CPI) improvement for the 6502 and its subset was 21%. The leading fetch BAM was not improved, but the trailing fetch BAM was.

Viper appears to be a plausible vehicle for rapid prototyping non-pipelined processors. At least it can quickly generate nontrivial designs of moderate quality.

The underlying hardware components available to the system are important. The differences in BAM performance, for example, between the different register file implementations, are as significant as synthesis optimizations.

Specification structure is important. The differences in BAM performance between the different fetch orders, for example, are as significant as synthesis optimizations.

Extensibility, which provides generality, is hard to do well in combination with optimization, which can be context specific (see Appendix C on adding operators and functional units).

A subsidiary conclusion of this work is that Prolog proved to be an effective implementation vehicle for Viper, excellent in expressive power and adequate in performance (see Appendix E).

Viper was successfully extended to support pipelined designs. One such design was synthesized, and exhibited impressive cost-performance results. Such designs are, however, not easily generated, because, effectively, the user must schedule all instructions by hand, guaranteeing that all will require the same number of pipeline stages.

4. Extensions to Viper

A number of possible extensions to the current Viper system are possible.

4.1. Extensions to Common Case Allocation

Several extensions to and variations on common case allocation can be explored. Such extensions and variations are probably best explored with hardware specifications for devices other than microprocessors (featuring more complex expressions).

One possibility would be to partition the global area constraint into required and optional components. These constraints could also additionally or alternatively be subdivided by type (such as registers and ALUs). This partitioning would alleviate the globally greedy behavior of a single constraint.

Another possibility would be to develop different metrics for functional unit selection. In the current version of Viper the most frequently used optional units are allocated first. Other possible metrics for choosing a unit to add could be: size (smallest first, which will ultimately add the greatest number of units), and generality (most functions supported, which might have the greatest utility).

A variation of utility constrained allocation would be to allocate resources for some percentile of instructions, rather than for some individual threshold percent. For example, allocate resources for the instructions that are used 90% or 95% of the time, rather than for individual instructions that are used more than 1% of the time. This could be done by keeping a running total of the frequencies of the instructions implemented up to that point, and allocate optional components up to a total cutoff frequency.

True data flow analysis for referencing values, and subsequent error detection or automatic temporary generation, is necessary. Currently the tracking of values in registers, and the detection of the need for temporaries and their creation, (see brk and jsr in the 6502) is done manually.

The allocator could be more intelligent about register ports. It could determine how many were needed and assign them, rather than requiring user declarations and references.

More generally, other operator-by-operator allocators could be examined (such as EMUCS [EMUCS]), and modified to use common case allocation.

4.2. Extensions to Hardware Implementation

The hardware generated by Viper can be made more complex and sophisticated, thereby increasing the opportunities for optimization (increasing also the difficulty and complexity of the optimization process).

First, Viper could better support more complicated expressions by treating expression temporaries in a more sophisticated manner, potentially evaluating an expression over multiple clock cycles. This would require interposing latches in the data path to hold such intermediate temporary values. This in turn would require more complicated methods of computing clock frequency and dealing with delay constraints.

Second, Viper could produce better designs by supporting more sophisticated finite state machine implementations, and making choices among those implementations. Examples are: PLA versus simple ROM versus microsubroutines. To do this adequately, low level estimates (of control speed and size, for example) would interact with high level decisions (such as the inlining of subroutines).

Third, Viper could be improved to generate more sophisticated pipelined designs, performing data path driven scheduling (see Chapter 13), taking into account the number of register ports, saving and restoring the pipeline (for interrupts), dealing with register file data dependencies, and using the frequency of branch instructions to control pipeline length.

4.3. Specification Extensions

In general terms, the system context (including caches and memory interfaces) needs to be specified. Viper currently provides just a simple memory interface.

Operator and functional unit extensibility needs to be examined and improved. It is currently rather difficult to add new operators and functional units (see Appendix C). The process can be better parameterized and more table driven.

For pipelined designs, it should be possible to specify pipelining details, such as delay slots after instructions, and instruction annulling.

5. The Challenges to High-Level Synthesis

In the long run, high-level hardware synthesis must address several issues if it is to produce high quality designs, and be of use to hardware designers. High-level synthesis efforts have traditionally been focused on scheduling and allocation. They are necessary parts of synthesis, and they are well-suited for algorithmic description and analysis. But they are not the only difficult or time-consuming aspects of hardware design. Other issues need to be solved.

- Real hardware is usually described at multiple levels of detail. Some parameters and features (such as pin specifications and timing waveforms) are low level. Synthesis systems must be able to effectively handle such detail. The problem is similar to the one facing the implementors of the Ada programming language, who must support hard real-time systems with performance requirements in the context of a high-level language.
- A total hardware system solution is needed. In particular, a contemporary microprocessor consists of much more than an integer unit core. Caches and memory systems are the focus of current microprocessor design. Board and system level issues also need to be addressed. Effectively synthesizing a microprocessor from an ISA requires all of these.
- High-level specification form is important (see, for example, the effects of the placement of instruction fetch in the BAM specification in Chapter 12). Even inter-block transformations like trace scheduling are dependent on specification form. Interactively applied, correctness preserving transforms, such as those used in the System Architect's Workbench ([SAW]), are necessary.
- Target hardware is important (see, for example, the effects of different register files in the BAM designs of Chapter 12). In particular, available control path implementations, register files, and functional units affect the quality of designs. Synthesis systems should support multiple implementations, and pursue, or let the user pursue, alternatives.
- Design management, software engineering paradigms, and logic synthesis may obviate much of the need for true high-level synthesis. Two fundamental reasons for using high-level synthesis are faster design times and designs that are correct by construction. But with the right tools, more traditional methods of design can be both speeded up and made less error prone.

On the other hand, the growing popularity of the VHDL language presents an opportunity. Synthesizable subsets of it are being defined, with associated tools ([VSS], for example). Synthesis capabilities will be more broadly available, and designers will become more aware of high-level synthesis support.

Bibliography

[ADAM]
'Experience with the ADAM Synthesis System'; Rajiv Jain, Kayhan Kucukcakar, Mitchell J. Mlinar, Alice C. Parker; *26th Design Automation Conference, 1989*; pp. 56-61.

[ADAM-Intro]
'A Design Utility Manager: the ADAM Planning Engine'; David W. Knapp, Alice C. Parker; *23rd Design Automation Conference, 1986*; pp. 48-54.

[ALERT]
'Methods Used in an Automatic Logic Design Generator (ALERT)'; Theodore D. Friedman, Sih-Chin Yang; *IEEE Transactions on Computers*; July 1969; pp. 593-614.

[APARTY]
'Architectural Partitioning for System Level Design'; E. Dirkes Lagnese, D.E. Thomas; *26th Design Automation Conference, 1989*; pp. 62-67.

[Aquarius]
'Aquarius -- A High Performance Computing System for Symbolic/Numeric Applications'; A.M. Despain, Y.N. Patt; *COMPCON 85*.

[Arch-Hayes]
Computer Architecture and Organization, Second Edition; John P. Hayes; McGraw-Hill, 1988.

[Arch-H&P]
Computer Architecture: A Quantative Approach; John L. Hennessy, David A. Patterson; Morgan Kaufmann, 1990.

[Arch-Kogge]
The Architecture of Pipelined Computers; Peter M. Kogge; Hemisphere (Taylor & Francis), 1981.

[Arch-Tanenbaum]
Structured Computer Organization, Third Edition; Andrew S. Tanenbaum; Prentice-Hall, 1990.

[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.

[Aunt]
'Aunt'; Peter Reintjes; *Fourth Symposium on Logic Programming*, September 1987.

[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.

[BECOME]
'BECOME: Behavior Level Circuit Synthesis Based On Structure Mapping'; Ruey-Sing Wei, Steven Rothweiler, Jing-Yang Jou; *25th Design Automation Conference, 1988*; pp. 409-414.

[Bridge]
'Bridge: A Versatile Behavioral Synthesis System'; Chia-Jeng Tseng, Ruey-Sing Wei, Steven G. Rothweiler, Michael M. Tong, Ajoy K. Bose; *25th Design Automation Conference, 1988*; pp. 415-420.

[BUD]
'Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions'; Michael C. McFarland; *23rd Design Automation Conference, 1986*; pp. 474-480.

[A2S] 'Reevaluating the Design Space for Register-Transfer Hardware Synthesis'; M.C. McFarland; *IEEE International Conference on Computer Aided Design, 1987*; pp. 262-265.

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

[CADDY]
'Synthesizing Circuits from Behavioral Level Specifications'; W. Rosenstiel, R. Camposano; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*; 1985; pp. 391-403.

[Cathedral]
'An Efficient Microcode Compiler for Application Specific DSP Processors'; Gert Goossens, Jan Rabaey, Joos Vandewalle, Hugo De Man; *IEEE Transactions on Computer-Aided Design*; September 1990; pp. 925-937.

[CHARM]
'A Global Dynamic Register Allocation and Binding for a Data Path Synthesis System'; Nam-Sung Woo; *27th Design Automation Conference, 1990*; pp. 505-510.

[Chippe]
'An Expert-System Paradigm for Design'; Forrest D. Brewer, Daniel D. Gajski; *23rd Design Automation Conference, 1986*; pp. 62-68.

[Chippe-Micro]
'State Synthesis and Connectivity Binding for Microarchitecture Compilation'; Barry M. Pangrle, Daniel D. Gajski; *IEEE International Conference on Computer Aided Design, 1986*; pp. 210-213.

[Chippe-Splicer]
'Splicer: A Heuristic Approach to Connectivity Binding'; Barry M. Pangrle; *25th Design Automation Conference, 1988*; pp. 536-541.

[CHS]
'Design Considerations for a Prolog Silicon Compiler'; Patrick C. McGeer et alia; November 1985.

[CMU-BLT]
'Behavioral Level Transformation in the CMU-DA System'; Robert A. Walker, Donald E. Thomas; *20th Design Automation Conference, 1983*; pp. 788-789.

[CMU-CAD]
'The CMU Design Automation System: An Example of Automated Data Path Design'; A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, J. Kim; *16th Design Automation Conference, 1979*; pp. 73-80.

[CMU-Cluster]
'Computer-Aided Partitioning of Behavioral Hardware Descriptions'; Michael C. McFarland; *20th Design Automation Conference, 1983*; pp. 472-478.

[CMU-DA]
'Automatic Data Path Synthesis'; Donald E. Thomas, Charles Y. Hitchcock, Thaddeus J. Kowalski, Jayanth V. Rajan, Robert A. Walker; *IEEE Computer*; December 1983; pp. 59-70.

[CMU-DR]
'A Model of Design Representation and Synthesis'; R.A. Walker, D.E. Thomas; *22nd Design Automation Conference, 1985*; pp. 453-459.

[CMU-DRT]
'Design Representation and Transformation in the System Architect's Workbench'; R.A. Walker, D.E. Thomas; *IEEE International Conference on Computer Aided Design, 1987*; pp. 166-169.

[CORAL]
'Linking the Behavioral and Structural Domains of Representation in a Synthesis System'; R.L. Blackburn, D.E. Thomas; *22nd Design Automation Conference, 1985*; pp. 374-380.

[CORALII]
'CORAL II: Linking Behavior and Structure in an IC Design System'; Robert L. Blackburn, Donald E. Thomas, Patti M. Koenig; *25th Design Automation Conference, 1988*; pp. 529-535.

[CSSP]
'Synthesis of Optimal Clocking Schemes'; Nohbyung Park, Alice Parker; *22nd Design Automation Conference, 1985*; pp. 489-495.

[DAA]
An Artificial Intelligence Approach to VLSI Design; Thaddeus J. Kowalski; Kluwer Academic Publishers, 1985.

[DAA-Proto]
'The VLSI Design Automation Assistant: Prototype System'; T.J. Kowalski and D.E. Thomas; *20th Design Automation Conference, 1983*; pp. 479-483.

[DAA-Intro]
'The VLSI Design Automation Assistant: What's in a Knowledge Base'; T.J. Kowalski and D.E. Thomas; *22nd Design Automation Conference, 1985*; pp. 252-258.

[DFBS]
'Global Hardware Synthesis from Behavioral Dataflow Descriptions'; Josef Scheichenzuber, Werner Grass, Ulrich Lauther, Sabine Maerz; *27th Design Automation Conference, 1990*; pp. 456-461.

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

[Emerald]
'Emerald: A Bus Style Designer'; C.-J. Tseng, D.P. Siewiorek; *21st Design Automation Conference, 1984*; pp. 315-321.

[EMUCS]
'A Method of Automatic Data Path Synthesis'; Charles Y. Hitchcock III, Donald E. Thomas; *20th Design Automation Conference, 1983*; pp. 484-489.

[Facet]
'Facet: A Procedure for the Automated Synthesis of Digital Systems'; Chia-Jeng Tseng and Daniel P. Siewiorek; *20th Design Automation Conference, 1983*; pp. 490-496.

[Flamel]
'Flamel: A High-Level Hardware Compiler'; Howard Trickey; *IEEE Transactions on Computer-Aided Design*; March 1987; pp. 259-269.

[Fred] 'An Object-Oriented, Procedural Database for VLSI Chip Planning'; Wayne Wolf; *23rd Design Automation Conference, 1986*; pp. 744-751.

[Graph]
'Synthesis Techniques for Digital System Design'; R. Camposano; *22nd Design Automation Conference, 1985*; pp. 475-481.

[HAGGLER]
'A Novel Approach to the Synthesis of Practical Datapath Architectures Using Artificial Intelligence Techniques'; N.S.H. Brooks, R.J. Mack; *1988 IEEE International Conference on Computer Design*, pp. 388-391.

[HAL]
'HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis'; P.G. Paulin, J.P. Knight, E.F. Girczyc; *23rd Design Automation Conference, 1986*; pp. 263-270.

[HAL-FDS]
'Force-Directed Scheduling in Automatic Data Path Synthesis'; P.G. Paulin, J.P. Knight; *24th Design Automation Conference, 1987*; pp. 195-202.

[HAL-FDLS]
'Scheduling and Binding Algorithms for High-Level Synthesis'; Pierre G. Paulin, John P. Knight; *26th Design Automation Conference, 1989*; pp. 1-6.

[HERCULES]
'HERCULES - A System for High-Level Synthesis'; Giovanni De Micheli, David C. Ku; *25th Design Automation Conference, 1988*; pp. 483-488.

[HIS] 'Synthesis Using Path-Based Scheduling: Algorithms and Exercises'; Raul Camposano, Reinaldo A. Bergamaschi; *27th Design Automation Conference, 1990*; pp. 450-455.

[HIS-DP]
'Area and Performance Optimizations in Path-Based Scheduling'; Reinaldo A. Bergamaschi, Raul Camposano, Michael Payer; *28th Design Automation Conference, 1991*; pp. 450-455.

[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.

[ITL] 'Synthesis and Optimization of Interface Transducer Logic'; Gaetano Borriello, Randy H. Katz; *IEEE International Conference on Computer Aided Design, 1987*; pp. 274-277.

[LPS] 'A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic'; Louis J. Hafer, Alice C. Parker; *IEEE Transactions on Computer-Aided Design*; January 1983; pp. 4-17.

[LYRA]
'Data Path Allocation Based on Bipartite Weighted Matching'; Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, Yu-Chin Hsu; *27th Design Automation Conference, 1990*; pp. 499-504.

[MacPitts]
'MacPitts: An Approach to Silicon Compilation'; Jay R. Southard; *IEEE Computer*; December 1983; pp. 74-82

[MacPitts-Intro]
'Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions'; Jeffrey Mark Siskind, Jay Roger Southard, Kenneth Walter Crouch; *1982 Conference on Advanced Research in VLSI; pp. 28-39.*

[MacPitts-Report]
'An Introduction to MacPitts'; J.R. Southard; *MIT Lincoln Laboratory Project Report RVLSI-3*, February 1983.

[MAHA]
'MAHA: A Program for Datapath Synthesis'; Alice C. Parker, Jorge T. Pizarro, Mitch Mlinar; *23rd Design Automation Conference, 1986*; pp. 461-466.

[MILO]
'MILO: A Microarchitecture and Logic Optimizer'; Nels Vander Zanden, Daniel Gajski; *25th Design Automation Conference, 1988*; pp. 403-408.

[MIMOLA]
'A New Synthesis Algorithm for the MIMOLA Software System'; Peter Marwedel; *23rd Design Automation Conference, 1986*; pp. 271-277.

[MIPS]
MIPS RISC Architecture; Gerry Kane; Prentice-Hall; 1988.

[NP] *Computers and Intractability*; Michael R. Garey, David S. Johnson; W.H. Freeman, 1979.

[Objects-Intensions]
'Objects as Intensions'; Weidong Chen, David Scott Warren; *Fifth International Logic Programming Conference and Symposium*; August 1988; pp. 404-419.

[Objects-Logical]
'Logical Objects'; John S. Conery; *Fifth International Logic Programming Conference and Symposium*; August 1988; pp. 420-434.

[OCCAM]
'OCCAM to CMOS: Experimental Logic Design Support System'; T. Mano, F. Maruyama, K. Hayashi, T. Kakuda, N. Kawato, T. Uehara; *Computer Hardware Description Languages and their Applications (CHDL 85 Proceedings)*, 1985; pp. 381-390.

[Oct] 'Data Management and Graphics Editing in the Berkeley Design Environment'; D. Harrison, P. Moore, R. Spickelmier, A.R. Newton; *Proceedings of the IEEE International Conference on CAD*, November 1986.

[PLEST]
'PLEST: A Program for Area Estimation of VLSI Integrated Circuits'; Fadi J. Kurdahi, Alice C. Parker; *23rd Design Automation Conference, 1986*; pp. 467-473.

[Preditor]
'A VLSI Design Environment in Prolog'; Peter Reintjes; *Fifth International Logic Programming Conference and Symposium*; August 1988; pp. 70-81.

[Prolog]
Programming in Prolog; W.F. Clocksin, C.S. Mellish; Springer-Verlag, 1981.

[Prolog-DA]
'Logic Programming and Digital Circuit Analysis'; W.F. Clocksin; *Journal of Logic Programming*; March 1987; pp. 59-82.

[Prolog-HDL]
'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.

[RLEXT]
'An Interactive Tool for Register-Level Structure Optimization'; David W. Knapp; *26th Design Automation Conference, 1989*; pp. 598-601.

[SAW]
Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench; D.E. Tho-

mas, E.M. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan, R.L. Blackburn; Kluwer Academic Publishers, 1990.

[SAW-Intro]
'The System Architect's Workbench'; D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn; *25th Design Automation Conference, 1988*; pp. 337-343.

[Sehwa]
'Sehwa: A Program for Synthesis of Pipelines'; Nohbyung Park, Alice C. Parker; *23rd Design Automation Conference, 1986*; pp. 454-460.

[Silc] 'The Silc Silicon Compiler: Language and Features'; T. Blackman, J. Fox, and C. Rosebrugh; *22nd Design Automation Conference, 1985*; pp. 232-237.

[SiliComp]
Silicon Compilation; Daniel D. Gajski, editor; Addison-Wesley, 1988.

[SLIMOS]
'Module Selection for Pipelined Synthesis'; Rajiv Jain, Alice Parker, Nohbyung Park; *25th Design Automation Conference, 1988*; pp. 542-547.

[SPARC]
SPARC RISC User's Guide; Cypress Semiconductor; 1990.

[SPEC]
SPEC Fact Sheet; Systems Performance Evaluation Cooperative, Fremont, CA; 1989.

[SUGAR]
'Synthesis by Delayed Binding of Decisions' J.V. Rajan, D.E. Thomas; *22nd Design Automation Conference, 1985*; pp. 367-373.

[Suite]
'A Prolog Benchmark Suite for Aquarius'; R. Haygood; *UC Berkeley CS Technical Report UCB/CSD 89/509*, April 1989.

[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.

[Trace-Carlson]
'The Bottom-Up Design of a Prolog Architecture'; Richard Carlson; *UC Berkeley CS Technical Report UCB/CSD 89/536*, May 1989.

[Tutorial]
'Tutorial on High-Level Synthesis'; Michael C. McFarland, Alice C. Parker, Raul Camposano; *25th Design Automation Conference, 1988*; pp. 330-336.

[Two-Dim]
'Algorithms for Hardware Allocation in Data Path Synthesis'; Srinivas Devadas, A. Richard Newton; *1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1987, pp 526-531.

[USC-AT]
'Predicting Area-Time Tradeoffs for Pipelined Design'; Rajiv Jain, Alice Parker, Nohbyung Park; *24th Design Automation Conference, 1987*; pp. 35-41.

[USC-DDS]
'A General Methodology for Synthesis and Verification of Register Transfer Designs'; Alice C. Parker, Fadi Kurdahi, Mitch Mlinar; *21st Design Automation Conference, 1984*; pp. 329-335.

[USC-Interface]
'Representation of Control and Timing Behavior with Applications to Interface Synthesis'; Sally A. Hayati, Alice C. Parker, John J. Granacki; *1988 IEEE International Conference on Computer Design*, pp. 382-387.

[USC-Overview]
'The ADAM Advanced Design Automation System: Overview, Planner, and Natural Language Interface'; John Granacki, David Knapp, Alice Parker; *22nd Design Automation Conference, 1985*; pp. 727-730.

[V] 'The V Compiler: Automating Hardware Design'; Viktors Berstis; *IEEE Design and Test*; April 1989; pp. 8-17.

[VSS] 'Synthesis from VHDL'; Joseph S. Lis, Daniel D. Gajski; *1988 IEEE International Conference on Computer Design*, pp. 378-381.

[VSS-VHDL]
'VHDL Synthesis Using Structured Modeling'; Joseph S. Lis, Daniel D. Gajski; *26th Design Automation Conference, 1989*; pp. 606-609.

[VSS-PS]
'Percolation Based Synthesis'; Roni Potasman, Joseph Lis, Daniel D. Gajski; *27th Design Automation Conference, 1990*; pp. 444-449.

[Viper]
A Prolog-Based Hardware Synthesis System: Source Code and Examples; William R. Bush; May 1992.

[Y-Chart]
'New VLSI Tools'; Daniel D. Gajski, Robert H. Kuhn; *IEEE Computer*; December 1983; pp. 11-14.

[YSC]
'Structural Synthesis in the Yorktown Silicon Compiler'; R. Camposano; *VLSI 87*; pp. 29-40.

[YSC-Design]
'Design Process Model in the Yorktown Silicon Compiler'; Raul Camposano; *25th Design Automation Conference, 1988*; pp. 489-494.

[YSC-Partition]
'Partitioning Before Logic Synthesis'; R. Camposano, R.K. Brayton; *IEEE International Conference on Computer Aided Design, 1987*; pp. 324-326.

Appendix A: SM1 Variants

This appendix presents simple machine specifications other than the SM1.

1. The SM2

The SM2 is an enhanced version of the SM1, complex enough to run Aquarius Prolog. It has two general purpose registers and an IO register. It has four additional instructions: subtract, register to register move, no-op, and branch on tag.

```
% SM2 -- two register SM1
stateRegister(x, 16).
stateRegister(y, 16).
stateRegister(pc, 16).
stateRegister(memAR, 16).
stateRegister(memDR, 16).
stateRegister(io, 16, output).
stateField(x, xtag, (15-12)).
stateField(x, xdata, (11-0)).
stateField(y, ytag, (15-12)).
stateField(y, ydata, (11-0)).
stateField(memDR,  opcode, (15-12)).
stateField(memDR,  address, (11-0)).

run :-
        write('--fetch '),stateCount(C1),write(C1),nl,
    fetch, !,
        write('--update '),stateCount(C2),write(C2),nl,
    stateUpdate, !,
        write('--access'),nl,
    access(memDR, opcode, OP), !,
        write('--execute '),write(OP),nl,
    !, execute(OP), !,
        write('--update '),stateCount(C3),write(C3),nl,
    stateUpdate, !,
        write('--recurse'),nl,
    run.
run :- true.

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

execute(addx) :- !,
    access(x, X), access(y, Y), T is X+Y, set(x, T).
execute(addy) :- !,
    access(x, X), access(y, Y), T is X+Y, set(y, T).
execute(subx) :- !,
    access(x, X), access(y, Y), T is X-Y, set(x, T).
execute(suby) :- !,
    access(x, X), access(y, Y), T is X-Y, set(y, T).
execute(andx) :- !,
    access(x, X), access(y, Y), T is X/\Y, set(x, T).
execute(andy) :- !,
    access(x, X), access(y, Y), T is X/\Y, set(y, T).
```

```
execute(movx) :- !,
    access(y, T), set(x, T).
execute(movy) :- !,
    access(x, T), set(y, T).execute(shrx) :- !,
    access(x, X), T is X>>1, set(x, T).
execute(shry) :- !,
    access(y, Y), T is Y>>1, set(y, T).
execute(loadx) :- !,
    access(memDR, address, M), set(memAR, M),
    mem_read,
    access(memDR, T), set(x, T).
execute(storx) :- !,
    access(memDR, address, M), set(memAR, M),
    access(x, T), set(memDR, T),
    mem_write.
execute(loady) :- !,
    access(memDR, address, M), set(memAR, M),
    mem_read,
    access(memDR, T), set(y, T).
execute(story) :- !,
    access(memDR, address, M), set(memAR, M),
    access(y, T), set(memDR, T),
    mem_write.

execute(jump) :- !,
    access(memDR, address, M), set(pc, M).
execute(nop) :- !,
    true.
execute(brnx) :-
    access(x, X), X<0, !,
    access(memDR, address, M), set(pc, M).
execute(brnx) :- !,
    true.
execute(brny) :-
    access(y, Y), Y<0, !,
    access(memDR, address, M), set(pc, M).
execute(brny) :- !,
    true.
execute(brvx) :-
    access(x, tag, T), T =:= var, !,
    access(memDR, address, M), set(pc, M).
execute(brvx) :- !,
    true.
execute(brlx) :-
    access(y, tag, T), T =:= list, !,
    access(memDR, address, M), set(pc, M).
execute(brlx) :- !,
    true.
execute(brvy) :-
    access(y, tag, T), T =:= var, !,
    access(memDR, address, M), set(pc, M).
execute(brvy) :- !,
    true.
execute(brly) :-
    access(y, tag, T), T =:= list, !,
    access(memDR, address, M), set(pc, M).
execute(brly) :- !,
    true.
```

```
execute(outx) :- !,
     access(x, X), set(io, X).
execute(outy) :- !,
     access(y, Y), set(io, Y).
execute(halt) :- !,
     fail.
```

2. The SM1 with Multi-Port Memory

The multi-port memory version of the SM1 includes two memory address and data registers. The first memory port is used for instructions, the second for data. Because there are no memory conflicts, two memory ports are not in fact needed.

The multi-port memory SM1 declarations are

```
stateRegister(ac, 16).
stateRegister(pc, 16).
stateRegister(memAR(1), 16).
stateRegister(memDR(1), 16).
stateRegister(memAR(2), 16).
stateRegister(memDR(2), 16).
```

and the fetch procedure is

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

The execute procedures are like their SM1 counterparts, but use port two.

3. The Pipelined SM1

This section presents the specification of the pipelined SM1, described in Chapter 13, and associated resource usage data.

The pipelined version of the SM1 differs from the basic version in four ways.

- It uses two port memory, the first port for instructions and the second for data (allowing parallel accesses).
- It uses the memory address register of the first port as the program counter (saving a register and the associated accesses).
- It employs delay operators to normalize instructions (see Chapter 13).
- It has a no-op instruction for delayed branches.

```
% pipelined SM1, two port memory
%   4 cycle
%   1: fetch from memory and pc increment
%   2: execute operand setup
%   3: execute memory operation
%   4: execute ALU operation
%   1 delay slot (PC changed in cycle 2)
%   2 preceding nops required for brn (AC changed in cycle 5)
%   2 preceding nops required for stor (AC changed in cycle 5)
%   case (execute) and cond (brn) take no cycles
%   (they are subsumed in pipeline control)

stateRegister(ac, 16).
stateRegister(memAR(1), 16).
stateRegister(memDR(1), 16).
stateField(memDR(1), opcode, (15-10)).
stateField(memDR(1), address, (9-0)).
```

```
stateRegister(memAR(2), 16).
stateRegister(memDR(2), 16).

run :-
    fetch,
    access(memDR(1), opcode, OP),
    execute(OP),
    run.
run :- true.

fetch :-
    mem_read(1),
    access(memAR(1), PC), P1 is PC+1, set(memAR(1), P1).

execute(halt) :- !,
    fail.
execute(add) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    mem_read(2),
    access(memDR(2), T), access(ac, AC), A is T+AC, set(ac, A).
execute(and) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    mem_read(2),
    access(memDR(2), T), access(ac, AC), A is T/\AC, set(ac, A).
execute(shr) :- !,
        delay,
        delay,
    access(ac, AC), A is AC>>1, set(ac, A).
execute(load) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    mem_read(2),
    access(memDR(2), T), set(ac, T).
execute(stor) :- !,
    access(memDR(1), address, X), set(memAR(2), X),
    access(ac, T), set(memDR(2), T),
    mem_write(2),
        delay.
execute(jump) :- !,
    access(memDR(1), address, T), set(memAR(1), T),
        delay,
        delay.
execute(brn) :-
    access(ac, AC), AC<0, !,
    access(memDR(1), address, T), set(memAR(1), T),
        delay,
        delay.
execute(brn) :- !,
        delay,
        delay,
        delay,
    true.
execute(nop) :- !,
        delay,
        delay,
        delay,
    true.
```

The following resource usage patterns have been simplified somewhat from the form presented in Chapter 13, in an attempt to make them more readable. The frequency of each set of equivalent instructions is given, followed, for each cycle, by the functional units used and the registers read and written. Note that the first pattern is tagged [none]. This block is always executed (it is instruction fetch).

```
[none] 1
    1 -mem(1)-ALU
        R: memAR(1)
        W: memAR(1)
[add,and] 0.25974
    2
        R: field(memDR(1),address)
        W: memAR(2)
    3 -mem(2)
    4 -ALU
        R: ac, memDR(2)
        W: ac
[shr] 0.038961
    2
    3
    4 -SHF
        R: ac
        W: ac
[load] 0.12987
    2
        R: field(memDR(1),address)
        W: memAR(2)
    3 -mem(2)
    4
        R: memDR(2)
        W: ac
[stor] 0.064935
    2
        R: ac, field(memDR(1),address)
        W: memAR(2), memDR(2)
    3 -mem(2)
    4
[jump] 0.0324675
    2
        R: field(memDR(1),address)
        W: memAR(1)
    3
    4
[brn-true] 0.0701298
    2
        R: field(memDR(1),address)
        W: memAR(1)
    3
    4
[brn-false] 0.0467532
    2
    3
    4
[nop] 0.279221
    2
    3
    4
```

Appendix B: Instruction Frequency Data

This appendix describes how SPEC benchmark results obtained for a different architecture, the SPARC, were mapped to the SM1 and the 6502. The overall approach was to identify equivalent instructions, categories of instructions, and operands, and use the corresponding counts.

The appendix also presents BAM instruction counts.

1. The SM1

The SM1 percentages were developed by hand, based on instruction categories. These percentages and categories are similar to those for the subset 6502.

```
% 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).
```

2. The Subset 6502

These percentages were also developed by hand, and are also based on instruction categories. The SPARC instruction categories used are noted in comments. Note that this is just one of three (gcc, eqntot, and spice) sets of frequencies.

```
% instructions (from cc)
count(adc,23.1). % all adds and subs
count(lda,20.7). % all loads
count(bmi,16.4). % all conditional branches
count(and,13.1). % all logicals
count(sta,7.7).  % all stores
count(lsr,3.7).  % all shifts
count(jsr,2.1).  % subroutine call plus save
count(rts,2.1).  % subroutine call plus restore
count(jmp,1.2).  % branch always
count(nop,0.1).
count(rti,0.1).  % rett
% addressing modes
%  (using load)
count(abs,56.4).  % rX + Y
count(imm,30.0).  % rX + g0
count(ind,13.0).  % rX + rY
count(zerop,2.7). % rX + 0 (equivalent to rX + g0)
% SPARC zerop equivalent, 13 bit immediate,
%   appears in non-UNIX system code
% interrupts
count(true,1).
count(int,0.01).
count(reset,0.001).
count(halt,0.001).
```

3. The Complete 6502

The complete 6502 implements a substantial number of instructions and addressing modes. Rather than convert all the data by hand, for three separate benchmarks, a program was written that performs the conversion automatically. It prompts for and accumulates the appropriate SPARC data, and generates corresponding 6502 percentages. Some marginal 6502 instructions have predefined low percentages.

A complete listing of the translation program can be found in [Viper].

4. The BAM Benchmark Composite

As mentioned in Chapter 9, a composite of 25 Prolog benchmarks, executing 50,283,248 instructions in total, was used in synthesizing the BAM. The specific benchmarks and instruction counts are give below.

Name	Description	Number of Instructions Executed
log10	Symbolic differentiation	851
ops8	Symbolic differentiation	901
times10	Symbolic differentiation	1185
divide10	Symbolic differentiation	1401
nreverse	Naive reverse of a 30-element list	7375
qsort	Quicksort of a 50-element list	8835
serialise	Calculate serial numbers of a list	12609
query	Query a static database (with integer arithmetic)	130080
boyer	An extract from a Boyer-Moore theorem prover	22211221
browse	Build and query a database	20256394
chat	Parse a set of English sentences	3796408
zebra	A logical puzzle based on constraints	2058133
tak	Recursive integer arithmetic	1415394
sendmore	-	1304335
poly10	Symbolically raise a polynomial to the tenth power	1285629
reducer	A graph reducer based on combinators	1087531
8queens	Solve the eight queens puzzle	937425
nand	A logic synthesis program based on heuristic search	466892
metaqsort	A meta-interpreter running qsort	123049
crypt	Solve a simple cryptarithmetic puzzle	111147
prover	A simple theorem prover	28007
fastmu	An optimized version of the mu-math prover	27017
mu	Prove a theorem of Hofstadter's mu-math	25058
flatten	Source transformation to remove disjunctions	19020
hanoi	Towers of Hanoi	8016

Table B-1: Benchmark Characteristics

The first eight benchmarks are the Warren benchmarks.

Instruction	Count
addi	8198725
ld	4571361
dref	4132158
ldd	3929985
st	2925280
jmp	2832117
cmp	2674685
bt	2581352
swt	2317063
jmpr	2144650
btgeq	1967558
pushdc	1916694
lea	1139850
xori	1061891
umax	965228
ldi	879325
push	832686
call	780329
pusht	779646
pushd	679300
cmpi	612444
ldx	612179
swb	445601
pop	390229
add	263981
std	166983
sti	140490
uni	82643
stdc	69509
slli	56944
andi	48979
and	38389
umin	18539
sub	15598
or	5394
srli	5388
stx	34
sra	21
sll	15
ori	1
srai	1
stid	1
srl	1
xor	1

Table B-2: Instruction Counts

Appendix C: Adding Operators and Functional Units

This section describes the process of adding operators (built-ins) and associated functional units to Viper, with specific reference to the BAM. The general process is important for two reasons. First, it affects the structure of Viper. Second, it is of some practical consequence, since adding operators and functional units is not uncommon, and hence should be relatively easy.

Summarizing the BAM additions monitored here, they are:

- new, complex comparisons involving tagged data (for cmp and cmpi);
- an additional test for unbound tags (for swb);
- arithmetic shift (for sra and srai);
- unsigned maximum and minimum (for umax and umin); and
- register file support.

In the following, note that more was required than making new entries in a library. Changes were made to the Prolog translator (tran), the allocator (sched, needs, and decl), and the trace scheduler (comi, com, and como), as well as to the library (lib and opmap).

1. Value Producing Operators

Three value producing operators -- operators used in expressions to the right of *is* -- were added for BAM -- arithmetic shift (ash), unsigned minimum (min), and unsigned maximum (max). These operators have, in Prolog specifications, the form

```
ShiftedQuantity is ash(Quantity, Shift), ...
Minimum is min(Value1, Value2), ...
Maximum is max(Value1, Value2), ...
```

Translation. A clause was added for each operator to recognize it for error detection purposes (in scanNumeric).

Library. Library entries (in opmap and lib) were made for asr-shf, min-cmp, and max-cmp (in these pairs, the first element is the operation and the second is the functional unit type).

Trace scheduling. In output conversion, clauses were added to recognize these operators as components of expressions (in convertRTSource, which handles embedded expressions).

2. Conditional Operators

Two simple conditional operators were added, not equal (=\=) and an unbound tag test (tagvar), which succeeds only when its first argument is unbound.

In Prolog specifications these operators have the form

```
Value1 =\= Value2, !,
tagvar(Tag1, Tag2), !,
```

Translation. A clause was added for each operator to recognize it as a conditional operator -- as a goal that might legitimately fail (in scanGoal).

Library. Library entries in opmap and lib were made for ne-cmp and tagvar-cmpvar.

Trace scheduling. In input conversion, a clause was added to map tagvar to a standard Prolog operator (==) understood by the trace scheduler (in comOp). In output conversion, clauses were added to recognize =\= and tagvar as control expressions (in convertSTExp), and a clause was added to map == back to tagvar (in convertRT).

3. Complex Operators

One complex operator was added, tval. This operator takes as one of its inputs a test to be performed (eq, ne, lts, ltu, tageq, int, etc.) on its other two inputs, and produces as its result a true-false bit (see [BAM-Manual], page 9). Since it has four arguments (three inputs and an output), it must be represented with two RTL transfers.

Specification. In Prolog specifications it has the form

```
tval(Test, A, B, Result), set(psw, tf, Result), ...
```

It would naturally be represented in RTL as

```
rtran(ID, Block, Test, A, B, tval, Result).
```

if a transfer could have four operands, but is instead realized in two register transfers

```
rtran(ID, Block, A, B, tval, Temp).
rtran(ID, Block, Temp, Test, tagcmp, Result).
```

where Temp is an RTL temporary (not a latch or register), and the tagcmp operator is a second, artificial operator always tied to tval.

Translation. A clause was added to recognize tval and create the extra tagcmp transfer (in scanGoal, modeled after four operand addc and subc).

Library. Library entries were made for tagcmp-cmptag; the second, tagcmp operator was used as the library key.

Allocation. A clause was added to recognize tval (in contrast to tagcmp) as an unneeded operator (in needsOperator). Another clause was added to recognize tval as an unscheduled operator (in declBindMap).

Trace scheduling. In output conversion, a clause was added to recognize tval as a component of the tagcmp expression (in convertRTSource).

4. Register File Operators

Two register file Operators were added, rval and wval, which provide single cycle access to registers in register files.

Specification. In Prolog specifications they have the form

```
rval(RegisterFilePort, RegisterIndex, Value), ...
wval(RegisterFilePort, RegisterIndex, Value), ...
```

Translation. A clause was added for each, to recognize it and to create the necessary data flow information about the port (in scanGoal).

Library. Mapping entries were made for rval-regfile and wval-regfile (in opmap only).

Allocation. Two clauses were added (in needsOperator) to recognize the operators as part of the register file, not requiring independent allocation. Two other clauses were added to treat rval and wval as data moves, not computations (in declBindEnable).

Trace scheduling. In output conversion, a clause was added to recognize rval as a component of expressions (in convertRTSource).

5. Register File Mechanics

Additional support was needed for register file ports and two cycle accesses. In particular, the port and register (reg) structures below, for example,

```
% 1 cycle access
rval(port(Register, Port), RegisterIndex, Value), ...
% 2 cycle access
set(port(Register, Port), RegisterIndex),
access(reg(Register, Port), Value), ...
```

required additions.

Allocation. Clauses were added (to schedResource and schedUse) to treat ports and regs as depending on the same resource, for two cycle access scheduling purposes. Clauses were also added (in needsOperand) to decompose ports and regs and recognize the needed register files. An additional clause was added (in declGetWidth) to get register file sizes from stateRegisterSet declarations.

Trace scheduling. Clauses were added (in src_use and dest_use) to schedule ports and regs properly (through dependencies). In output conversion, a clause was added to recognize reg(Name, Port) as a register (in convertIsReg).

6. Summary

Adding the above eight operators involved considerably more than new entries in the functional unit and operator mapping data bases. Seven additional modules (tran, comi, com, como, sched, needs, and decl) required modifications, adding 31 clauses in 16 procedures.

Some of these changes are to be expected; the register file models provide substantially different functionality, and more than half of the additions (17) support those models.

Another class of predictable changes involves syntax. It is understandable that adding new operators will require added syntactic processing (including translation between normal and trace scheduled RTL formats). Again, more than half of the additions (18) involve syntax.

Nonetheless, this extension process is relatively complicated and delicate for a system user, an architect, to embark on. Prolog's pattern matching style aids the process, such as it is, somewhat. It is clear, however, that Viper could be more parameterized, and the process simplified.

A further consideration, not addressed here, is the addition of simulation support, both at the Prolog and RTL levels.

Appendix D: The ASP System

This appendix describes the levels of hardware description in the ASP system, and outlines the operation of the components at the various levels.

1. ASP Overview

The ASP system is based on two central principles of implementation. First, the hardware synthesis process has been decomposed into many discrete layers, each of which encapsulates a specific, narrow body of expertise. Second, that expertise has been represented as both rules and algorithms, as appropriate.

We have adopted the first principle because we need to understand the hardware design and synthesis process well in order to automate it, and we can only understand it if we can divide up the overall process into manageable, discrete subproblems. The general paradigm we follow is hierarchical, with each layer making incremental design decisions that are added to the accumulating knowledge about the design. When, during system design, the implementation of a given layer has been unclear, it has been subdivided into further component layers.

The disadvantage to this approach is that it is hierarchical, encountering the problem that many high-level design decisions are made using low-level knowledge (an observation verified in [BUD]). Our solution to this problem is to use accurate estimators whenever possible, and to constrain the high-level problem domain to single-chip microprocessors. Ultimately, when all the individual levels are sophisticated enough, we may develop design iteration techniques that allow humans or an expert system planner to improve designs.

We have adopted the second principle because of the basic nature of many CAD problems, and because our implementation vehicle, Prolog, encourages it. Many CAD problems are computationally difficult, being NP-complete, and algorithmic approximation techniques, such as simulated annealing, are useful only for a limited subset of problems. Data path allocation, for example, is frequently done heuristically (as is done in [DAA] and [HAL]). On the other hand, much of hardware synthesis involves translation from one representation to another, and much of this translation involves simple bookkeeping that can best be done in an algorithmic form (this issue was discussed in [DAA]). Furthermore, good algorithmic techniques exist for some problems (compaction, for example), and yet they can be augmented with useful heuristics in key places (for, say, dealing with diagonal constraints). Another system implemented in Prolog ([OCCAM]) has also used a mixed approach.

Thus far we have found that our two principles have made the full-range synthesis problem tractable. The hierarchical approach has been advantageous for four reasons. First, a primary goal of ASP is to automate the synthesis process as much as possible; human interaction and, to a lesser extent, redesign, are less important. Second, incremental redesign can be avoided by more global analysis and more accurate estimators at higher levels. Third, a layered approach is more flexible in that there are no global data structures and thus no global ramifications to representation changes, and preserving that flexibility is still desirable. Fourth, the hierarchical approach and its limitations are worth exploring further.

2. Decomposition of Silicon Compilation

A full behavior-to-silicon compiler is a complex undertaking. We decompose the silicon compilation problem into four major abstract problem domains, ordered hierarchically.

The top level of our system is the behavioral domain. This level generates a data path (a set of functional units), controlled by a finite state machine, from an input specification written in Prolog. Both standard compiler techniques and hardware-specific knowledge are used in this process. The ASP component performing this task is called Viper.

The second level is the logic domain. The purpose of this domain is to present the behavioral component with abstract components, and to map those components to boolean structures. This level synthesizes and connects the finite state machine and functional units generated by the behavioral level. This level encompasses the traditional tasks of state assignment and logic synthesis.

The third level is the circuit or functional domain, which includes placement, routing, and control generation. This level consists of placing and connecting transistors, and operates on sticks-and-elements objects.

The fourth level is the geometric domain. This level involves the generation of design-rule-correct geometry. This domain encompasses the tasks of compaction and device-level simulation.

Clearly there is some interaction between the levels. No layout generator can ignore the constraints inherent in technology, such as, for example, the richer connectivity of two layers of metal. Similarly, the data path allocator can only use known functional units.

3. Specification: Executable Prolog

The highest level of specification in ASP is executable Prolog. This level is discussed further in Chapter 4.

4. High-Level Synthesis

Viper is the high-level synthesis component in the ASP system, discussed in the body of this work.

5. Specification: Data Paths and Control Paths

This specification level is the symbolic output of high-level synthesis. It is the level of traditional functional level simulation. It defines data paths and associated finite state machines.

Data paths are defined in terms of functional unit types (such as registers and ALUs) and netlists connecting them (buses and control lines). Functional unit declarations have the form

```
element(<type>, <name>,
    [<input-bus-netlist>], [<output-bus-netlist>],
    [<control-signal-netlist>]).
```

where *<type>* is the type of some functional unit known to the library. For example, an accumulator register (the sign bit of which is used for negative value tests) and an ALU could be declared with

```
element(reg,ac,[bus(3)],[bus(1)],[ac(sign)]).
element(alu,alu(1),[port(bus(1),1),port(bus(2),2)],[bus(3)]).
```

Finite state machines are defined in terms of states, where each state has the form

```
state(<state-name>, [<list-of-actions>], <next-state>).

<action> ::= move(<src>, <bus>, <dst>) | enable(<element>, <function>)
<next-state> ::= <state-name> | switch(<value>, [<list-of-cases>])
<case> ::= case(<condition>, <state-name>)
```

For example, the first state transition adds the values in ac and memDR, and stores the result in ac. The second state transition simply selects the next state based on the opcode field of the memDR.

```
state(bc(2,3),
    [move(ac,bus(1),port(alu(1),1)),
        move(memDR,bus(2),port(alu(1),2)),
        move(alu(1),bus(3),ac),
        enable(alu(1),add)],
    bc(1,1)).

state(bc(1,3),
    [],
    switch(field(memDR,opcode),
        [case(add,bc(2,1)),
         case(sub,bc(3,1)),
         case(load,bc(4,1)),
         case(stor,bc(5,1))
         ...])).
```

Note that states (bc(1,1)), buses (bus(1)), and functional units (alu(1)) are named using Prolog structures rather that simple atoms (such as bus1 and alu1).

6. Boolean Synthesis

Boolean synthesis involves the realization of boolean structures. This is the elaboration, down to the bit level, of constructs generated at the behavioral level. The finite state machine is realized through the generation of PLA equations, and the data path is completely specified, including all bit widths, control lines, buses, multiplexers, and drivers.

Specifically, boolean synthesis consists of five steps.

- Explicit bus multiplexers and tri-state drivers are created. This requires new element declarations and modified netlists, as well as modified state definitions (move actions are converted to enable the multiplexers and drivers).
- The data path is expanded to include functional unit bit widths and to refer explicitly to bitwise control lines.
- Ports and pads are declared.
- State assignment is done.
- The equations for a PLA are generated. (Optionally, a ROM can be generated.)

Significant opportunities exist for improving this level. State assignment, PLA equation generation, and ROM generation can all be optimized.

This level did not exist separately in the ASP prototype system ([ASP-Prototype]), in which behavioral and boolean synthesis were combined. In the current system they have been split, which clarifies the separate issues raised at each level.

7. Specification: PLA

ASP PLAs have an AND plane and an OR plane; input bit lines are fed to to the AND plane, the outputs of which are then connected to the OR plane, from which the output bit lines are taken. This level of specification is similar in general purpose to BLIF (1/0 logical function matrix).

These inputs and planes are straightforwardly represented with the following forms.

plaIn(<input-signal-name>).

plaAndOut(<and-signal-name>, [<list-of-inputs>]).
<input> ::= <input-signal-name>
| inv(<input-signal-name>)

plaOrOut(<output-signal-name>, [<list-of-and-signals>]).

plaAlias(<signal-name>, <output-signal-name>).

Note that inverted inputs are available to the AND plane. Note also the alias form, which is used for specifying that two output signal names are equivalent and are generated by the same or term.

For example, these fragments illustrate the use of the forms:

```
plaIn(state(0)).
plaIn(state(1)).
plaIn(state(2)).
plaIn(state(3)).
plaIn(ac(sign)).

plaAndOut(bc(3,1),
    [inv(state(0)),state(1),inv(state(2)),inv(state(3))]).
plaAndOut(test(ac(sign),bc(10,1)),
    [state(0),state(1),inv(state(2)),inv(state(3)),inv(ac(sign))]).

plaOrOut(state(3),
    [bc(3,3),bc(4,1),bc(4,2),bc(6,1),test(ac(sign),bc(10,1))]).

plaAlias(reg(ac,fn),state(3)).
```

8. Specification: Data Path Modules

Data path modules are elaborated forms of the elements generated by behavioral synthesis. All elements and all connectivity are fully specified.

Module declarations are similar to elements and have the form

module(<type>, <name>, <bit-range>,
<data-inputs>, <data-outputs>,
<control-inputs>, <control-outputs>).

where *<bit-range>* is the range of bits for which the module is defined.

A set of modules is arrayed into a data path during circuit synthesis. A data path consists of a rectangular structure with data buses running horizontally and control lines running vertically. Thus data and control lines are separated in module declarations.

For example, the above elements expand into

```
module(reg,ac,15-0,
    [bus(3)],[out(ac)],[reg(ac,fn),reg(ac,clock)],[ac(sign)]).
module(ts,ts(ac,bus(1)),15-0,
    [out(ac)],[bus(1)],[ts(ac,bus(1),fn)],[]).
module(alu,alu(1),15-0,
    [bus(1),bus(2)],[out(alu(1))],
    [alu(1,fn(1)),alu(1,fn(2))],[_]).
module(ts,ts(alu(1),bus(3)),15-0,
    [out(alu(1))],[bus(3)],[ts(alu(1),bus(3),fn)],[]).
```

The ts modules are tri-state drivers for the buses. Necessary control lines have also been added. All these modules are 16 bits wide.

The library of functional units currently consists of registers, latches, incrementers, ALUs, and shifters (see the discussion of libraries below and in Chapter 8).

9. Topological Circuit Synthesis

The input to this level is a collection of module definitions. The output is a sticks-based layout description. The data path generator creates individual bit slices and then arrays them.

Each bit slice is a strip of CMOS transistors. It consists of a horizontal row of P transistors and a row of N transistors, separated by a routing region, and bounded at the top and bottom by power and ground rails, which are shared with the next bit slice. Buses run horizontally through the bit slice in the routing region; control lines run vertically. Horizontal routing, including power, ground, and buses, is in metal-1; vertical routing is done in metal-2.

There are three major issues with this form of layout:

- how the modules in a bit slice should be ordered, with the goal being to minimize the intermodule routing in the channel;
- how the transistors within a module should be ordered, in order to maximize sharing of power and ground connections; and
- how to connect heterogeneous bit slices, where slices differ from one another and yet must abut.

In ASP, the module generator orders the bit slices using either the min-cut algorithm or simulated annealing; we are experimenting with the two methods. The Uehara-Van-Cleemput algorithm was used for transistor ordering, but we are currently using a depth first search. Heterogeneous bit slices are handled by creating a hypothetical union bit slice that contains all possible modules; this union bit slice is then ordered, and the resulting template is used to create each individual bit slice.

The input to the module generator as shown above consists of module names, types, and sizes, along with a two-dimensional netlist, with unique vertical names and generic horizontal names (which are instantiated bitwise by the module generator).

The module library consists of and-or-invert gate definitions of module functionality. In fact, the module generator consists of two distinct pieces; one orders modules and generates bit slices, and the other orders transistors within a module and generates module layout. The module library could define modules directly in terms of layout, and simply use the first part of the module generator.

The module generator in the ASP prototype system ([ASP-Prototype]) could not generate heterogeneous bit slices. This limitation required a new data path generator. The code would also not port from C-Prolog to Quintus Prolog.

10. Specification: Sticks

This specification level is composed of wires and elements, which in turn can be transistors and contacts. This level is referred to as Sticks In Prolog (SIP). In detail,

wire(<material>, <from-point>, <to-point>, <width>, <signal>).

trans(<type>, <source-point>, <gate-point>, <drain-point>,
<width>, <length>,
<source-signal>, <gate-signal>, <drain-signal>).

cont(<type>, <center>, <offset>, <signal>).
<offset> ::= e / n / w / s / none

pin(<side>, <location>, <layer>, <electrical-node>, <label>).
<side> ::= top / bottom / left / right

maxrow(<maximum-row-number>).
maxcol(<maximum-column-number>).

Pins are used to specify terminals on the cell, which is the collection of elements bounded by maxrow and maxcol.

For example, an inverter can be represented as

```
wire(m1,pt(0,0),pt(0,5),1,1).
wire(m1,pt(0,1),pt(2,1),1,1).
wire(m1,pt(10,0),pt(10,5),1,2).
wire(m1,pt(10,1),pt(8,1),1,2).
wire(m1,pt(8,3),pt(2,3),1,3).
wire(m1,pt(6,3),pt(6,5),1,3).
wire(p,pt(8,2),pt(2,2),1,in).
wire(p,pt(6,0),pt(6,2),1,in).
trans(nd,pt(2,1),pt(2,2),pt(2,3),4,2,1,4,3).
trans(pd,pt(8,1),pt(8,2),pt(8,3),2,2,2,4,3).
cont(m1nd,pt(2,1),na,1).
cont(m1nd,pt(2,3),na,3).
cont(m1pd,pt(8,1),na,2).
cont(m1pd,pt(8,3),na,3).
pin(top,(6,0),p,1,4).
pin(bottom,(6,5),m1,1,3).
maxrow(10).
maxcol(5).
```

SIP is also hierarchical. Cells can be composed of other named cells to any depth. The composition form is the cell, to wit

cell(<name>, <location>, <rotation>, <mirror>)

For example, four inverters can be arrayed with

```
cell(inv,pt(0,0),none,none).
cell(inv,pt(0,10),none,none).
cell(inv,pt(0,20),none,none).
cell(inv,pt(0,30),none,none).
```

When space information (see below) is added to SIP, CIF follows directly.

11. Geometric Synthesis

The Sticks-Pack environment consists of a technology independent compactor that creates spaced layout and simulation files from sticks-and-elements descriptions, a joiner that joins together cells generated by the compactor, a global placer, a global router, and a simulator that simulates sticks-based cells.

The compactor uses an algorithm similar to zone refining to perform a rough spacing of the elements. Floor and ceiling profiles for each layer of material are maintained. Elements from the ceiling are moved directly across the molten region to the floor, where spacing requirements are calculated, and diagonal constraints are noted. Rules then are used to shift the elements to better fit their environment.

Large layouts in Sticks-Pack are realized by joining small cells together. Leaf cells (cells of the lowest level consisting of transistors and wires) are compacted individually and constitute the building blocks for larger modules.

The global placer and router then place and route individual compacted blocks to produce the final layout. pp The output of Sticks-Pack is Space information (see below).

12. Specification: Space

This level defines physical location and, in conjunction with its associated SIP, is equivalent to CIF. It maps the virtual grid locations used by SIP to physical locations, using the following forms:

row(<virtual>, <physical>, <cell>).
col(<virtual>, <physical>, <cell>).
rowbound(<lowest-physical-row>, <highest-physical-row>, <cell>).
colbound(<lowest-physical-col>, <highest-physical-col>, <cell>).
maxrow(<virtual-max>, <cell>).
maxcol(<virtual-max>, <cell>).
hix(<physical-max>, <cell>).
hiy(<physical-max>, <cell>).

For example, the inverter presented above in SIP is compacted to

```
row(0,6,inv).
row(1,32,inv).
row(2,32,inv).
row(3,32,inv).
row(4,32,inv).
row(5,32,inv).
row(6,56,inv).
row(7,56,inv).
row(8,96,inv).
row(9,96,inv).
row(10,122,inv).
col(0,20,inv).
col(1,20,inv).
col(2,48,inv).
col(3,76,inv).
col(4,76,inv).
col(5,76,inv).
rowbound(6,122,inv).
colbound(20,76,inv).
maxrow(10,inv).
maxcol(5,inv).
hiy(96,inv).
hix(128,inv).
```

13. Representation Issues

An early design of the system ([CHS]) used a common unifying data structure; this approach was abandoned because the needs of and relationships between various synthesis components were poorly understood, and because the resulting common data base would have required too great an implementation effort.

One issue that we have bypassed is automated consistency -- from lower levels to higher and within a level. We assume correctness by construction. This is a problem when humans modify the design, or when low level information is passed up and redesign is done. This is a serious issue beyond our resources ([ADAM]). It also requires disciplined tool development ([CORALII]), which is difficult in our rapid prototyping environment.

14. The ASP Library

Elements in the ASP library are defined in terms of blocks of gates. The definition essentially consists of a netlist of blocks. For example, a two input multiplexer is defined by

```
signals(mux2,[input1(N),input2(N)],[output(N)],[control(N)],[]).
block(mux2,cbar(N),inv(control(N))).
block(mux2,outBar(N),
    aoi(nor(and(cbar(N),input1(N)),and(control(N),input2(N))))).
block(mux2,output(N),inv(outBar(N))).
```

The signals clause defines the input, output, and control signals. Each block clause specifies an output and a function for computing that output.

Appendix E: The Use of Prolog

This appendix discusses the advantages and disadvantages of Prolog in light of the Viper and ASP experiences.

1. Hardware Specification in Prolog

Clocksin has applied logic programming to the automated design of digital circuits [Prolog-DA]. He concentrated his effort on structural issues, rather than behavioral or geometrical ones, and considered the problems of circuit rewriting, gate assignment, determination of signal flow, and specialization of standard designs.

In contrast, the ASP system attacks the entire hardware synthesis problem, from executable behavioral specification to geometric layout. This problem in turn decomposes into a number of substantial subproblems, including simulation (at various levels of design), the allocation of hardware resources, the construction of finite state machines, the implementation of collections of boolean functions as topologically efficient circuits, the global placement and routing of various circuit blocks, and the effective compaction of two-dimensional layout.

Clocksin essentially dealt with the representation of connectivity. The ASP system deals with the representation of behavior, connectivity, and geometry. After considerable practical experience and the exploration of alternatives, the ASP system gravitated to a paradigm different from Clocksin's.

1.1. Representations of Connectivity

Clocksin identified three methods of representing connectivity, functional, extensional, and definitional. The functional method uses functional composition to build more complex functions out of more primitive ones. For example, the *and* of the *or* of two signals, a and b, with the inversion of a third, c, is represented as

```
and(or(a, b), invert(c)).
```

The extensional method uses the same ground terms in different clauses as wire names to connect the clauses. With this method the same example appears as

```
cell(or, [a, b], [t1]).
cell(invert, [c], [t2]).
cell(and, [t1, t2], [out]).
```

where cells consist of a type, a list of inputs, and a list of outputs. The definitional method uses Prolog variables as wire names, and composes functions by defining clauses with more primitive components appearing as goals. In this style the example has the form

```
fn(A, B, C, Out) :-
    or(A, B, T1),
    invert(C, T2),
    and(T1, T2, Out).
```

Clocksin prefers the definitional method, primarily because it is modular, hierarchical, and executable.

In general, the extensional method was chosen for ASP. It allows us to take advantage of the relational database inherent in Prolog. Function and representation are coupled with the definitional and functional methods; decoupling them with the extensional method makes it easier to explore and manipulate designs. The price paid for this decoupling is the loss of direct execution and Prolog backtracking. The price is not high, because simulation interpreters are easily constructed, and, in general, design space exploration requires more control than simple backtracking.

The following subsections review various ASP specification levels in the light of these concerns; the levels are introduced in Appendix D.

1.2. Behavioral Specification

It is possible to carry machine state in Prolog variables, passed from one recursive call to the next, rather than using register declarations and global state. This was done in the prototype ASP system; that version of Viper automatically translated such specifications into the form currently accepted by Viper as input. Such a pre-synthesis step could be added. The current input is at the lowest level of executable specification, reflecting the imminent realization of state as hardware registers.

1.3. RTL Specification

It would be possible to represent register transfers in a definitional form. Prolog variables could be used instead of symbolic register names, and the transfers could be grouped in blocks of goals. The disadvantage to such a representation is primarily with respect to synthesis. In scheduling, the transfers are scanned by index in order. For allocation they are scanned by index, but also scanned by operator and by operand. The relational power of Prolog is very useful here. Also, with respect to register names, allocation requires that registers be defined. The RTL simulator, which interprets transfers and transitions, was straightforward to construct.

1.4. Specification of Functional Unit Declaration and Use

It would certainly be possible to represent elements and states in a definitional form, replacing bus names with Prolog variables. Bus allocation is important and done at this stage, however, and an ancillary table relating element variables with allocated bus assignments would be necessary. The relational form is a simpler mechanism for synthesis.

The functional unit level defines the control signals needed to drive the functional units, which in turn represent the global state of the machine. This is a non-hierarchical, global level of description, to which the extensional method is well-suited.

1.5. PLA Specification

The flattened, tabular, extensional form used here closely approximates the arrayed form of the final PLA layout. It is similar to the input forms required by various available PLA optimization tools (not written in Prolog).

1.6. Data Path Module Specification

It would be possible to cast these modules in a definitional form, with the explicit connectivity replaced by Prolog variables. The basic conceptual problem with the definitional form in this context is that variables are dynamic connections reformed on each procedure call, whereas physical nets of connectivity are static, formed once when the hardware is created. It would be possible to analyze a definitional specification and bind net names to various variables, but, for synthesis, the static extensional form is more fundamental. Most CAD systems use the extensional form (see [Oct] and [Preditor], for example).

1.7. Specification of Sticks

The sticks and space levels are layer-based. They represent geometry, and are no longer executable. They are best stored relationally to simplify access, so that they can be scanned in different ways (such as all elements of a specific layer, or all elements at a specific point) depending on the circumstances.

1.8. Specification Summary

Other systems have used Prolog for hardware specification ([Prolog-DA], [Aunt]), but they have not used it in a full-range synthesis system. The ASP system covers many levels of abstraction, and the constructs needed at different levels vary considerably.

Nonetheless, the extensional method of representation has proved to be of general utility for synthesis, because of its relational properties, and because it is a natural way of representing static connectivity.

2. Experience with Prolog

The remarks that follow reflect the pragmatic experience of programming ASP and Viper in Prolog.

In this context, the good aspects of Prolog are that: it operates at a high level of abstraction, cleanly supporting pattern matching, relational data bases, and powerful and natural methods for iterating over and selecting objects (especially setof); it has a simple, powerful paradigm; and, for the sort of rapid prototyping done with Viper, Prolog strikes a good balance between speed of programming (fast) and speed of execution (adequate).

The bad aspects are: debugging -- the process of debugging in C-Prolog and SICStus is primitive compared to debugging in Smalltalk, for example; large systems are not easy to construct -- the global data base paradigm clashes with modularity requirements, and this clash has not been successfully resolved; and assert and retract are useful and not completely developed -- a method for localizing them (such as theories) would be both natural and lead to improved performance and safety.

Speed of execution has historically been an issue surrounding Prolog. It was in some circumstances an issue in the development of ASP.

It was not a problem with Viper, using either SICStus 0.6 or C-Prolog 1.5. Translating the base 6502 version into register transfers using C-Prolog, for example, took a little under 6 minutes on a Sun 3/50, and a little over a minute on a SPARCstation 1:

```
3/50: 342.4u 7.9s 7:04 82% 32:200k 36+17io 5pf+0w
SS1: 80.2u 2.8s 1:49 75% 0+712k 11+22io 10pf+0w
```

Speed was a problem with some of the lower level ASP tools, which had to deal with thousands of geometric elements, and were orders of magnitude slower than equivalent C programs. In particular, the ASP compactor was rewritten to use lists instead of assert and retract, in an effort to improve its performance.

Cell	Version 1	Version 2	Version 3	Version 4
a3.sip	127.13	215.23	-	176.87
action.sip	152.95	200.15	-	161.48
atgen.sip	155.44	207.32	-	164.38
bs.sip	9699.06	-	-	-
instLat.sip	103.72	142.15	-	141.60
inv.sip	5.30	23.52	-	19.90
lat.sip	131.92	175.43	-	147.85
passgate.sip	14.75	28.67	-	23.64
plaout.sip	6522.60	-	-	-
reg.sip	456.05	447.62	429.23	421.50
sm1pla.sip	11722.4	-	-	-
sm1reg.sip	912.69	1011.20	1020.51	-
smx.sip	3224.48	-	-	-

Table E-1: C-Prolog Runtimes

Cell	Version 1	Version 2	Version 3	Version 4
a3.sip	37.20	-	-	-
action.sip	37.42	-	-	-
atgen.sip	35.39	-	-	-
bs.sip	2349.80	1732.71	1234.52	-
instLat.sip	28.77	-	-	-
inv.sip	3.15	3.78	2.84	2.93
lat.sip	28.58	-	-	-
passgate.sip	5.12	-	-	-
plaout.sip	1406.93	950.97	639.15	-
reg.sip	99.80	99.54	81.13	-
sm1pla.sip	2325.05	1726.92	1190.63	-
sm1reg.sip	205.53	200.53	168.49	155.52
smx.sip	589.45	534.93	-	-

Table E-2: Quintus Prolog Runtimes

Version:

1: the original version, with many asserts and retracts;

2: some (Xdist, Ydist, and fence) asserts and retracts go to lists.

3: most asserts and all retracts go to lists.

4: lists go to structured difference lists.

The improvements realized through successive versions clearly improved performance, but both speed and memory usage were ultimately not competitive with equivalent tools written in C.