Co-Synthesis of Instruction Sets and Microarchitectures

Ing-Jer Huang

Advanced Computer Architecture Laboratory

ACAL-TR-94-10

August 1994

Keywords: application specific instruction set processor, instruction set design, resource allocation, code generation

Abstract

The design of an instruction set processor includes several related design tasks: instruction set design, microarchitecture design, and code generation. Although there have been automatic approaches for each individual task, the investigation of the interaction between these tasks still primarily relies on designers' experience and ingenuity. It is thus the goal of this research to develop formal models and algorithms to investigate such interaction systematically.

This dissertation presents a two-phase co-synthesis approach to the problem. In the architectural level, given a set of application benchmarks and a pipeline structure, the ASIA (Automatic Synthesis of Instruction set Architecture) design automation system generates an instruction set and allocates hardware resources which best fit the applications, and, at the same time, maps the applications to assembly code with the synthesized instruction set. This approach formulates the co-design problem as a modified scheduling/allocation problem. A simulated annealing algorithm is used to solve the problem. Following ASIA, the microarchitectural-level design automation system PIPER accepts the instruction set architecture specification and generates a pipelined microarchitecture which implements the instruction set, and a reordering table which guides the compiler backend (reorderer). This approach relies on an extended taxonomy of inter-instruction dependencies and the associated hardware/software resolutions.

The techniques are demonstrated with both illustrative and practical experiments. The results show that the techniques are capable of synthesizing instruction set processors that are as good as or better than the manually-designed instruction set architecture VLSI-BAM in application-specific environments, based on a design metric consisting of the instruction set size, cycle count and hardware resources. In addition, these techniques can be used to characterize architectural properties of application benchmarks.

CO-SYNTHESIS OF INSTRUCTION SETS AND MICROARCHITECTURES

by

Ing-Jer Huang

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering - Systems)

DISSERTATION COMMITTEE MEMBERS

Professor Alvin M. Despain

Professor Kai Hwang

Professor Peter Danzig

August 1994

Acknowledgments

I did not finish up this dissertation alone! It is by Lord's mercy that bountiful helps have accompanied me throughout the journey to the degree.

I would like to thank my committee chairman, Professor Alvin Despain, who has led me through the world of design automation for microprocessors. His continuous support and encouragement have been the essential fuel to my study. In addition, the Advanced Computer Architecture Laboratory (ACAL) has provided a unique environment where my vision has been broadened. I would like to thank my committee members, Professor Kai Hwang for his warm support, and Professor Peter Danzig for his humor and comments. I would also like to thank all of the members of ACAL. I have very much enjoyed many fruitful discussions with them, and especially the questions they raised in the dry runs for my presentations for conferences, workshops, the qualification examination and defense. These are the most challenging questions that I have ever had. I am grateful to Paul Suhler, Linda Maher and Kevin Obenland for reading and suggesting improvements to my dissertation draft.

It is the endless and unconditional support from my parents Chorng-Shyuan and Huei-Mei that I was able to come to the university to pursue my Ph.D. degree. They are always there for me whether I am high or low. The encouragements and constant prayers of my sisters I-Chun and I-Huei are the empowering supply for my study.

Special thanks go to my wife Shu-Ing, also a Ph.D. student but more than a student. She knows what to do at the right time. She took care of many other things so I could concentrate on the writing.

In addition, it is the lovely church life that has enriched my life and has kept me from being buried in all kinds of burdens.

Abstract

The design of an instruction set processor includes several related design tasks: instruction set design, microarchitecture design, and code generation. Although there have been automatic approaches for each individual task, the investigation of the interaction between these tasks still primarily relies on designers' experience and ingenuity. It is thus the goal of this research to develop formal models and algorithms to investigate such interaction systematically.

This dissertation presents a two-phase co-synthesis approach to the problem. In the architectural level, given a set of application benchmarks and a pipeline structure, the ASIA (Automatic Synthesis of Instruction set Architecture) design automation system generates an instruction set and allocates hardware resources which best fit the applications, and, at the same time, maps the applications to assembly code with the synthesized instruction set. This approach formulates the co-design problem as a modified scheduling/allocation problem. A simulated annealing algorithm is used to solve the problem. Following ASIA, the microarchitectural-level design automation system PIPER accepts the instruction set architecture specification and generates a pipelined microarchitecture which implements the instruction set, and a reordering table which guides the compiler backend (reorderer). This approach relies on an extended taxonomy of inter-instruction dependencies and the associated hardware/software resolutions.

The techniques are demonstrated with both illustrative and practical experiments. The results show that the techniques are capable of synthesizing instruction set processors that are as good as or better than the manually-designed instruction set architecture VLSI-BAM in application-specific environments, based on a design metric consisting of the instruction set size, cycle count and hardware resources. In addition, these techniques can be used to characterize architectural properties of application benchmarks.

Chapter 1 Introduction

1.1 Motivation

An *instruction set architecture* (ISA), or *instruction set* (IS), is an abstract representation of an instruction set processors (ISP's). It is the interface between hardware and software (Figure 1.1). It characterizes the organization and functionality of hardware which are visible to software such as compilers and operating system kernels. It also serves as the behavior specification that hardware designers follow when designing the microarchitecture of a computer. The *microarchitecture*[1] (MA) or *organization* refers to the structural aspect of a computer's design, such as functional units, the register file, the memory system, the interconnect structure and the pipeline configuration. Microarchitectures and instruction sets are interdependent entities in the following senses.

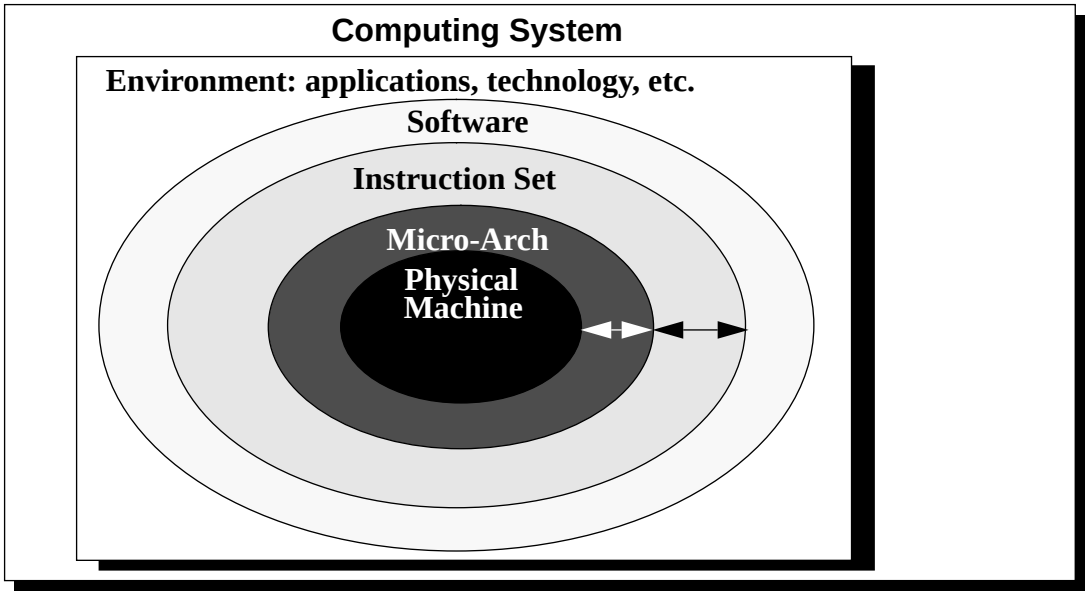


Figure 1.1 The role of the instruction set and microarchitecture

1. The term *microarchitecture* is used by some to denote a class of micro-programmed hardware organization, for example in [19], whereas some others, for example in [32], use it in a more general sense such as the one employed in this text.

1. On one hand, a microarchitecture provides the micro-operations which implement an instruction set; on the other hand, an instruction set is an encoding of micro-operations available in a microarchitecture.

2. The implementation of the microarchitecture determines the clock cycle time and gives the hardware cost such as silicon area, power consumption and the number of I/O pins. In compiling an application program written in a high level language down to machine code, the design of the instruction set determines how efficiently the underlying microarchitecture matches the application. And thus, together with the clock cycle time, determines the performance. Therefore, the microarchitecture defines the hardware costs of the instruction set, while the instruction set defines the efficiency of the microarchitecture.

3. The problem of designing microarchitectures and instruction sets is a typical chicken-or-egg problem: they mutually depend on each other. Whether an instruction can be implemented depends on the supporting microarchitecture. On the other hand, whether a microarchitecture feature should be included depends on the instruction set to be implemented. In practice, this problem is dealt with in an iterative manner as shown in Figure 1.2.

   For example, a designer may first assume an initial set of instructions, and then constructs an initial microarchitecture under some design constraints. Then with some possible alteration to this initial microarchitecture the designer checks if instructions can be added or deleted, also under some design constraints. He then repeats this process until he reaches an equilibrium point or runs out of design time. In practice, other design approaches are also employed.

4. For a given instruction set, various microarchitectures are possible, which include different pipeline organizations and circuit modules with different performance/ cost tradeoffs. A scheduling or reordering phase in the compiler backend is often necessary to ensure that the original sequential semantics of the software programs can be preserved. This phase also optimizes the assembly code for the program to be executed on the particular microarchitecture implementation. On the other hand, the complexities of the compilation techniques and the application environments may constrain the design space of the microarchitecture. For example, a highly pipelined microarchitecture, which requires sophisticated scheduling techniques to generated efficient assembly code, may be feasible in an embedded system design. In this case, the software programs are compiled (possibly by hand) and permanently loaded into the memory. They are never changed and they are executed repeatedly for the life of the hardware system. However, a highly pipelined microarchitecture may be infeasible for the purpose of software development in which compilation happens very frequently, and short compilation time is a major advantage.

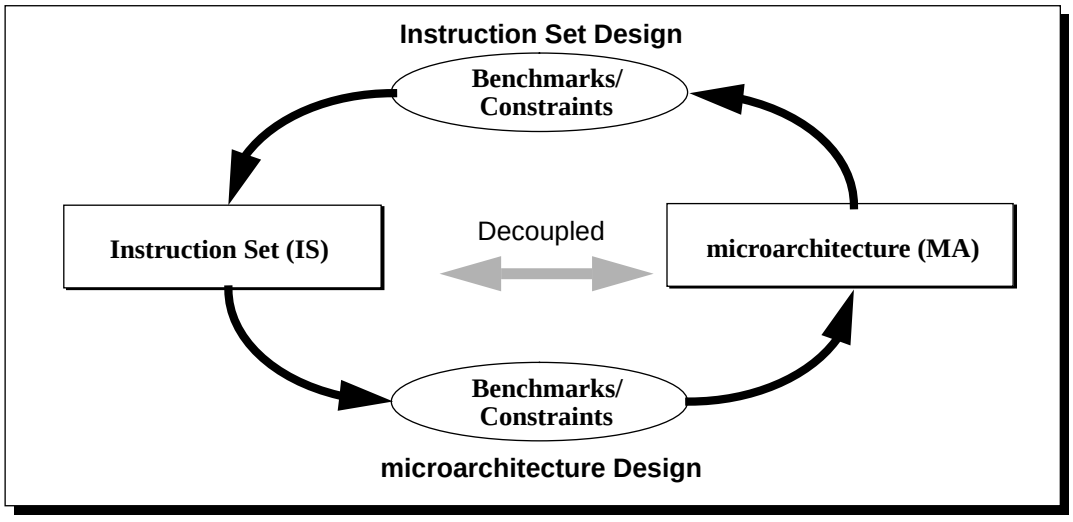


Figure 1.2 Iterative approach to designs of IS and MA

5. The performance of the instruction set and microarchitecture varies with the application domain. Also the characteristics of the expected application domain may affect the equilibrium point between the microarchitecture and instruction set. Therefore, in most ISA designs, a set of benchmark programs, which is believed to be the representative of the expected application domain, is employed to evaluate the design of the microarchitecture and the instruction set. This is shown in Figure 1.2. For example, such a design example can be found in [25]. Several benchmark sets such as SPEC [73], Livermore Loops [58], Whestone [17] and the Prolog benchmark suite [30] have been proposed to evaluate computer ISA's.

Therefore, when designing a good instruction set processor, features in the instruction set, microarchitecture and the compiler backend are chosen under cost constraints. These design options have to be tuned toward the representative application benchmarks and the intended application environment. Currently, most designs of instruction set processors are carried out manually. The balance between various design options relies mainly on the designers' experience and skills. Such a manual design process is slow, requiring much labor by smart people, and many iterations of analysis and redesign occur.

Computer-aided design (CAD) tools become necessary to facilitate the design process of instruction set processors and improve the understanding of the interaction among their software/hardware components. Two branches of CAD tool development are related to instruction set processor design. High level (behavioral) synthesis for instruction set processors corresponds to the microarchitecture design phase in Figure 1.2. This approach accepts a given instruction set architecture specification and generates the microarchitecture implementation at the register transfer level (RTL). [7]

and [15] are examples of such an approach. The design space of this approach is constrained by the given instruction set specification. On the other hand, design automation systems such as [5] and [36] address the instruction set design phase in Figure 1.2. These systems accept a set of application benchmarks, a given microarchitecture and instruction format constraints, and produce a best instruction set according to the given objective function. However, which microarchitecture will yield the best instruction set is left to the designer of the microarchitecture.

These two kinds of tools can be used in the iterative approach of Figure 1.2 for the design of instruction set processors. However, there are two major limitations with this iterative approach. First, the tools are not designed with the interaction in mind as ineffective interfaces may make iterations difficult. Second, iterative approaches may be computationally inefficient.

These limitations raise the following research question:

> *How can the best combination of instruction set and microarchitecture be derived from application benchmarks under a given objective function?*

Here the application benchmarks are given in an intermediate form (IR) similar to the intermediate code generated by compiler front-ends. The objective function specifies how the designer desires to trade off competing features such as hardware resources, cycle count, static code size, and instruction set size.

1.2 The Thesis

What is the relationship between an application benchmark and the microarchitecture? The application benchmark can be viewed as a specification of a computing task to be performed on the target machine. At the microarchitectural level, the application benchmark is performed by executing a sequence of instructions. Each instruction con-

sists of a group of micro-operations. The micro-operations in the instruction sequence represent a scheduled instance of the dependency graphs of the micro-operations. The micro-operations manipulate the state of the processor. Therefore, the internal representation of the benchmark in the microarchitecture can be viewed as a sequence of processor state changes. The process of mapping the instruction sequence into state-change sequence can be viewed as a compilation and execution process, as shown in Figure 1.3 (a). Now the reverse of this compilation process can be viewed as instruction formation under a predefined hardware resource, as shown in Figure 1.3 (b). First, for each pair of state change, there is at least one set of dependency graphs of micro-operations that accomplishes the state change. Second, the micro-operations can be mapped into time

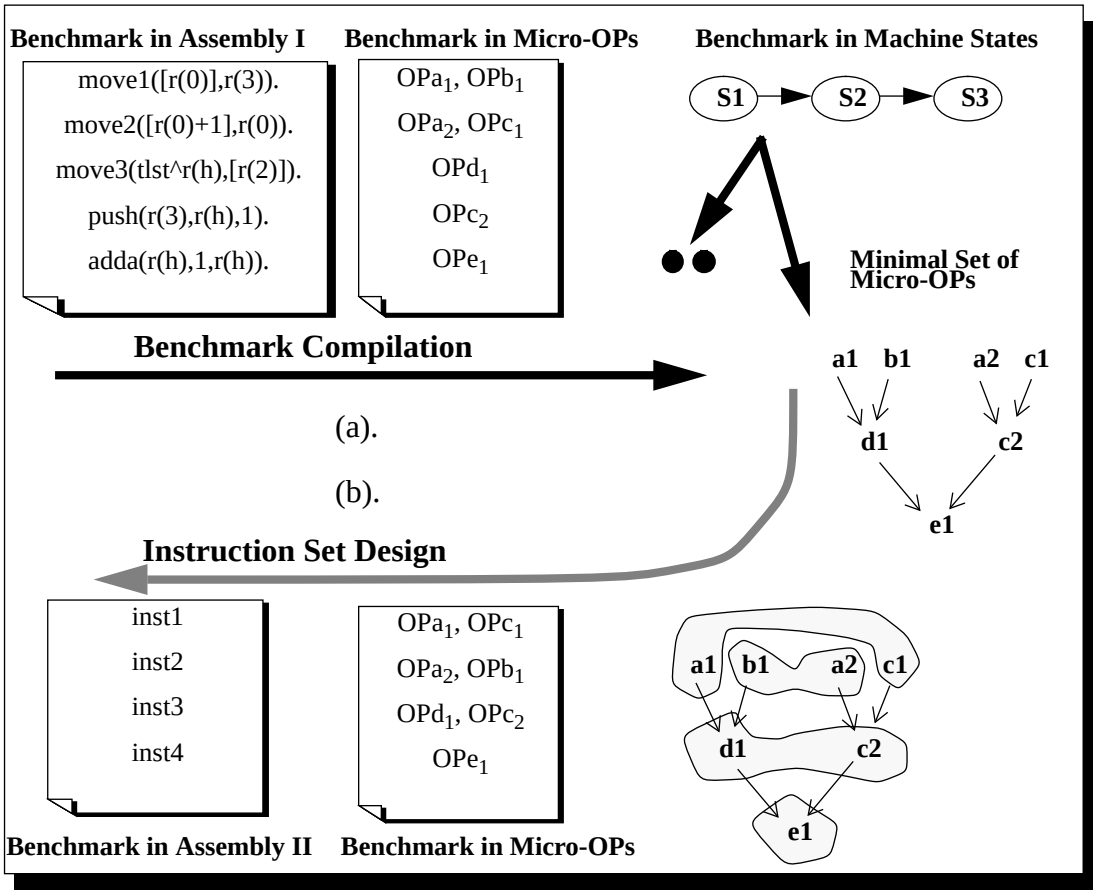


Figure 1.3 The machine code (instructions) and micro-operations for a benchmark

steps. micro-operations mapped to the same time step can then be considered as an instruction. The collection of the time steps becomes the assembly code. Hence the instruction set design (instruction formation) becomes a constrained scheduling problem of the micro-operations. The constraints are the predefined hardware resource, the dependency graph, and the size of each step in the schedule (representing the instruction word length). The optimization is performed upon several competing factors including the number of steps in the schedule (cycle count), and number of unique steps in the schedule (size of instruction set). The problem can be further generalized by allowing the hardware resources to be varied. Now the simultaneous instruction set and microarchitecture design can be described as a simultaneous scheduling and allocation problem with an objective function controlling the trade-off.

Therefore, given a set of primitive operations (micro-operations) which represent possible operations in microarchitectures, the application benchmarks can be mapped into dependency graphs of these primitive operations. From these dependency graphs, the instruction set and microarchitecture can be simultaneously derived through a combined scheduling and allocation process.

Modelling the design problem as a scheduling and allocation problem provide two advantages: first, techniques that have been developed for high level synthesis and language compilers can be employed; second, known methods of scheduling and allocation help in understanding and exploiting the design of instruction set processors.

1.3 Research Scope, Direction, and Limitation

Most design automation efforts dedicated to instruction set processor design can be characterized into the following categories[2]:

2. Detailed descriptions for the classification are given in Section 2.2 on page 17.

- **Instruction Set Synthesis (ISS)**. Assuming or given a model of instruction sets, this approach generates an instruction set that optimizes an objective function such as instruction set size, cycle count, cycle time, transistor count, etc. The models of the data path and control path are either explicitly expressed as the design constraints, or are embedded into the model of the instruction set.
- **Instruction Set Mapping (ISM)** or **Code Generation (CG)**: This approach addresses the problem of systematically mapping the given application into assembly code with the given instruction set. This approach usually involves the design of a retargetable code generator or mapper.
- **Microarchitecture Synthesis (MS)**: Given an instruction set specification, this approach synthesizes a microarchitecture at the register transfer level (RTL), usually including both data path and control path, which implements the given instruction set.

As we have seen in Section 1.1 that these problems are interdependent ones. It is the goal of this research to express these problems as an integrated one, in order to understand the principles which govern the interactions and optimal solutions of these problems.

Based on the scheduling/allocation problem formulation, this dissertation presents an approach to the integrated problem at the architectural and microarchitectural levels. Two integrated design automation systems have been built to address problems in the two levels. At the architectural level, given a set of application benchmarks and a pipeline structure, the ASIA (Automatic Synthesis of Instruction set Architecture) design automation system generates an instruction set and allocates hardware resources which best fit the applications, and, at the same time, maps the applications to assembly code with the synthesized instruction set. This phase addresses the synergy among instruction sets, instruction set mapping and hardware resources for a given pipeline structure.

At the next level (microarchitectural level), given an instruction set architecture specification which is synthesized by ASIA or other approaches, the PIPER design automation system generates a pipelined microarchitecture which implements the instruction set, and a reordering table which guides the compiler backend (reorderer). This approach addresses the synergy between pipeline structure and code generation for a given instruction set.

The limitations of this dissertation can be summarized as follows:

- Instruction set. The target instruction set is assumed to be of fixed word width which is given by the designer. Furthermore, the bit widths of instruction fields such as register index, tag value and immediate data are also given by the designer. The current research work does not attempt to determine the optimal widths for the fields, which is one of the important problems in instruction set architecture design.
- Microarchitecture. Ideally, it is desirable that the best combination of the instruction set, pipeline structure, and allocation of hardware resources can be determined at the same time. However, in order to manage the complexity of the problems, the problems are solved by keeping one fixed while optimizing for others. In the ASIA approach, the combination of the instruction set, resource allocation and code generation is determined by the system, while the pipeline stages are given by the designer. On the other hand, in the PIPER approach, the pipeline structure and resource allocation are determined by the system, while the semantics of the instruction set is given by the designer, only allowing the system to change the timing property of the instruction set. ASIA and PIPER can then be used in an iterative fashion to determine the best combination of the instruction set, pipeline structure and resource allocation.
- Code generation. ASIA synthesizes the best instruction set for the given application programs. Assembly code for the given application programs is automatically generated by the optimization process. However, if one decides to map other application

programs, which are not applied to synthesize the instruction set, to the synthesized instruction set, other techniques in code generation should be used, instead of ASIA. On the other hand, PIPER does not map the application programs to the pipelined microarchitecture it synthesizes. Instead, it generates an interface (reordering table) to direct the compiler backend (reorderer) in reordering the sequence of assembly code so as to preserve the sequential semantics of the code.

- Other concerns. This dissertation does not automatically generate instructions and hardware mechanisms for operating system support, interrupt/trap handling and I/O control, unless they are explicitly specified as part of the application programs and/or instruction set behavior. However, these can be handled by including a set of synthetic benchmark programs that contain these operations.

1.4 Research Contributions

The work in this dissertation provides integrated problem formulations to the combined problems of instruction set design, microarchitecture design and code generation. The integrated formulations help better understanding of their design processes and interactions. Based on the formulation, efficient algorithms are developed to make design automation system practical in synthesizing designs and exploring design trade-offs among competing factors in both hardware and software.

At the architectural level, ASIA can be used as a fast prototyping tool to determine feasible initial designs for new instruction set architecture design, and as a tool for the evaluation of existing instruction set architectures in application specific environment. Some case studies of ASIA's applications are presented in Section 8.1 on page 112, Section 8.2 on page 122 and Section 8.3 on page 129.

PIPER can be used to generate the register transfer level (RTL) implementation for the architecture synthesized by ASIA, providing more accurate performance/cost

measures which can be fed back to ASIA to guide the design process. In addition, PIPER can be used to explore the tradeoffs between microarchitecture and the complexity of the compiler backend. Furthermore, it can be used to extend the service life and improve the performance of existing instruction set architecture by simultaneously reimplementing hardware with pipeline structures and generating the interface to patch the original software environment so as to take advantage of the pipelined microarchitecture. Examples of PIPER's applications are given in Section 8.4 on page 141, including the synthesis of an industrial processor TDY-43.

1.5 Dissertation Outline

This dissertation is organized as follows. Chapter 1 presents the design problem and motivation. Chapter 2 reviews related work in automatic instruction set design, code generation and high level synthesis, and compares the research in this dissertation with the related work. Chapter 3 describes the global design flow of ASIA and PIPER. It also describes the framework of the Advanced Design Automation System (ADAS), a full range design automation system for microprocessors, into which both ASIA and PIPER are integrated.

Chapter 4 presents the models of instruction sets, microarchitectures and compiler backends used in this dissertation. Chapter 5 presents the problem formulation and algorithm for architectural problems addressed in ASIA: instruction set design and resource allocation.

Chapter 6 presents an extended taxonomy of inter-instruction dependency and their hardware and software resolutions. These serve as the key techniques for PIPER to simultaneously synthesize pipelined microarchitectures and interfaces to compiler backends and explore their tradeoffs. Chapter 7 presents algorithms which make use of the

taxonomy and resolutions discussed in Chapter 6 to synthesize the pipelined microarchitectures and compiler backend interfaces.

Chapter 8 demonstrates the techniques in this dissertation with a set of experiments. Chapter 9 draws conclusions and points out limitations of this dissertation and suggests future research directions. lists the references used in this dissertation.

Brief descriptions, exemplar input/out specifications and programs for every design phase of ASIA and PIPER are available for anonymous ftp at the host `zelea.usc.edu` under the directory `/pub/software/asp`. Please read the 'readme' file under the directory for further information.

Chapter 2 Background

In this chapter, research work in automating the design of instruction set processors is reviewed. We begin with a discussion of the design objects and metrics that are of common interest at the architectural and microarchitectural levels, then classify design automation for instruction set processors, based on the particular design objects interested, and follow with a review of related work in each category.

2.1 Design Objects And Metrics at Architectural and Microarchitectural Levels

The design objects are hardware and software components that the designers want to produce or control during the design process, including input specifications and constraints, output (implementation), and design metrics. The important design objects at the architectural or micro-architectural level are discussed below.

2.1.1 Hardware

- The *architecture model* specifies the architectural style of the processor, such as application-specific integrated circuits (ASIC), digital signal processor (DSP), application-specific instruction set processor (ASIP), general purpose instruction set processor (ISP), etc. The implementation style is usually also specified in the model to constrain the design space, such as uniprocessor, very-long-instruction-word processor (VLIW), pipeline, superscalar, and multiprocessor implementations.

- A *data path* is the data part of the microarchitecture which implements the given architecture model. The data path, consisting of registers, latches, functional units, interconnects, memory and I/O, provides mechanisms to manipulate the data. The basic design tasks for the data path include allocation of resources, assignment of operations to resources, and construction of interconnects.
- A *control path* is the control part of the microarchitecture. The control path controls the sequence of operations in the data path to obtain the desired output. The basic design tasks of the control path include selection of control styles, e.g., microprogrammed, finite state machine, counter, etc., allocation of control registers which store the control information, and generation of (symbolic) boolean equations and glue logic.
- *Chip area*, *cycle count*[1], *(clock) cycle time*, and *power dissipation* are attributes for a particular data/control path implementation. They also serve as design metrics to measure the quality of implementation and guide the design tradeoffs. The design metrics are usually obtained through modeling, except the cycle count, which can be obtained directly through compilation and simulation (execution). In addition, some design metrics at the architectural and microarchitectural levels are represented by symbolic values, instead of exact values, since low level implementation decisions have not yet been made. For example, the chip area is usually represented by the number of transistors or logic gates; the power dissipation is usually represented by the switching activities in the circuit.

1. The cycle count is the number of clock cycles taken to execute a given application.

2.1.2 Interface

- The *instruction semantics* refers to the behavior of the instruction, including micro-operations (MOPs) contained in the instruction, their operational relation (chaining, parallel, or sharing operands) and their timing relation. Whether an instruction semantics is feasible depends on the characteristics of the application, the architectural model, and the data/control path. Rich instruction semantics may require a complex architectural model and data/control path to support it.
- The *instruction format* refers to how the instruction word is partitioned into fields. The availability of an instruction format determines whether a proposed instruction semantics can be supported, and complexity of the data path. An instruction set with multiple instruction formats requires different ways of interpreting bits in the instruction register (IR). The same bit in the IR may belong to different fields in different instructions. Therefore, bit-wise manipulation on the instruction bus and proper interconnects may be necessary to support multiple instruction formats.
- The *instruction set size* is the number of valid instructions supported by the processor. The size has impact on the control path and the effectiveness of compiling applications to assembly code. The increase in the instruction set size may have the following effects: (1) the control logic also grows; (2) applications can be more effectively compiled into assembly code since more (possibly powerful) instructions are available for optimization; (3) the computational burden of the compiler also increases because it has to consider more instructions.

2.1.3 Software

- *Application*s are software programs which are selected to represent the typical computation tasks to be executed on the processor. The design of the processor is usually customized for the given application, especially in an application specific environ-

ment such as embedded systems. Applications may be given in either high level languages such as C, Fortran, Lisp, Prolog, etc., or intermediate code such as the three-address code commonly used in compilers, or other intermediate representations such as data/control flow graph, or dependency graphs of MOPs or register transfers (RTLs).

- *Assembly code* is the representation of the application at the instruction set level. The binary version of the assembly code (machine code) is the lowest representation of the application which the processor fetches from memory, decodes and executes.
- The *code generator* is usually the first component of the compiler backend which takes account of the detailed architectural features of the processor. A code generator maps machine-independent intermediate code to assembly code, based on the instruction set of the target processor. Code generators are machine-dependent, since processors have their own instruction sets which are different from each other. Even for processors supporting the same instruction set, but with different implementation technologies, various versions of code generators are used to deal with instructions with different performance and cost due to the variation of implementations. In order to save development efforts, a code generator can be made *retargetable* by dividing the generator into two parts: the machine-independent mapping algorithm, and the machine-dependent mapping table which specifies how each intermediate code is mapped to a sequence of target instructions. How to construct an efficient mapping table becomes part of the design task in designing architectures.
- The *peephole optimizer* is a component of the compiler backend which takes care of the microarchitecture feature of the processor. It takes in the assembly code generated by the code generator, searches for some patterns of instruction sequences, and replaces them with instruction sequences with better quality[2]. In addition to optimization, repairing the assembly code to satisfy some particular features of the microarchi-

tecture of the processor, e.g., branch/memory/operation delays, can be performed in the peephole optimizer as well. Similar to the code generator, the peephole optimizer is also machine-dependent. Therefore, it is usually divided into two parts: the machine-independent optimization algorithm and the machine-dependent pattern pairs. How to construct pattern pairs of high quality becomes part of the design challenges in designing microarchitecture.

- *Instruction simulators* are necessary to measure the cycle count of the benchmark programs executing on the target processor. There are usually two levels of instruction simulators: one for the architecture level with the assumption of sequential execution; one for the microarchitecture level where instruction level parallelism such as pipelining is exploited. The architectural level simulator measures performance in terms of the number of instructions executed. The microarchitectural level simulator measures performance in terms of the number of clock cycles executed. It is equal to the summation of the number of instructions executed, number of NOP instructions executed, and number of pipeline stall cycles due to branch/memory/operation delays, cache miss, etc. Both simulators are machine dependent, and have to be constructed for every processor designed.

2.2 Classification of Design Automation for Instruction Set Processors

The design of an instruction set processor is a complex task, with subtasks including the designs of hardware (processors), software (compiler backends) and interface (instruction sets). Most design automation approaches to instruction set processor design can be characterized into three categories: instruction set synthesis, code genera-

2. The better quality refers to one or more of the measurements: shorter instruction sequences, fewer registers used, fewer memory operations, fewer useless operations, etc. The actual selection of these measurements is based on the objective of the optimization given by the user.

tion, and high level synthesis. The difference among these approaches can be described in terms of the design objects and metrics, described in Section 2.1.

Table 2.2 on page 31 lists the design objects and metrics, and their relationships with each category of design automation approach. For each design object or metric, '++' indicates that it is a major input specification or constraint of the corresponding design automation approach; '- -' indicates that it is a major output; '+' and '-' indicate respectively the input and output that are either of secondary concern, or indirect cause/ consequence of the corresponding approach. An unmarked cell indicates that the design object or metrics is irrelevant to the corresponding approach.

2.2.1 Instruction Set Synthesis (ISS)

Assuming or given a model of instruction sets, this approach generates an instruction set that optimizes an objective function which consists of either abstract or concrete design metrics such as instruction set size, cycle count, cycle time, transistor count, and power dissipation. The models of the data path and control path are either explicitly expressed as the design constraints, or are embedded into the model of the instruction set.

2.2.2 Instruction Set Mapping (ISM) or Code Generation (CG)

This approach addresses the problem of systematically mapping the given application into assembly code with the given instruction set. This approach usually involves

Design Objects & Metrics	Instruction Set Synthesis* (ISS)		Code Generation (CG)		Microarchitecture Synthesis (MS)	
HARDWARE						
Architectural model	+ +	DSP, ASIP, general purpose ISP,...; the major input to ISS	+ +	DSP, ASIP, general purpose ISP,...; the major input to CG	+ +	DSP, ASIP, general purpose ISP,...; the major input to MS
Control path	-	determined by the size and semantics of the instruction set.			- -	the major output of MS; includes boolean equations for the control logic, state registers and interfaces to data path
Data path	+ +	# of architected registers, functional units, memory ports, I/O ports; data path topology	+ +	# of architected registers, functional units, memory ports, I/O ports; data path topology	- -	the major output of MS; inludes net lists of registers, functional units and interconnect components, and interfaces to control path
Chip area	–	mainly affected by control path (data path is fixed); most ISS either ignores this issue or models it with some abstract measure.			- -	the major objective MS minimizes or takes as a design constraints
Cycle count	- -	depends on how well the instruction set match the benchmark	- -	the major objective that CG minimizes	- -	the major objective MS minimizes or takes as a design constraints
Cycle time	-	difficult to estimate; not measured in most ISS's			- -	the major objective MS minimizes or takes as a design constraints
Power dissipation	-	depends on the instruction set and scheduling; however, most ISS does not consider the power issue.	-	depends on the sequence of instructions and types of computation in the sequence	- -	the major objective MS minimizes or takes as a design constraints
INTERFACE						
Instruction semantics	- -	ISS determines the behavior of the instructions	+ +	the major input to CG	+ +	part of the instruction set architecture specification; the behavior of instructions
Instruction format	- -	ISS determines the usage of instruction bits	+ +	the major input to CG	+ +	part of the instruction set architecture specification; the usage of instruction bits
Instruction set size	- -	ISS determines the number of instructions	+ +	the major input to CG	+ +	part of the instruction set architecture specification; the number of supported instructions

Table 2.1 The categories of design automation for instruction set processors and their design objects and metrics. Approaches that span more than one category such as the work in this dissertation are not listed in this table.

Design Objects & Metrics	Instruction Set Synthesis* (ISS)		Code Generation (CG)		Microarchitecture Synthesis (MS)	
SOFTWARE						
Application	+ +	given as a way to measure the effectiveness of the instruction set	+ +	CG takes in as input the intermediate representation of application programs		
Assembly code			- -	CG generates as ouput the assembly code (of the target processor) for the given application	+ +	given to MS to estimate the cycle count when executing the application on the synthesized processor
Code generator (CGR)	–	depends on the instruction set; however, most ISS's do not generate CGR.	–	CG may either generate assembly code directly, or indirectly by generating retargetable CGR and PO.		
Peephole optimizer (PO)	–	depends on the instruction set; however, most ISS's do not generate PO.	–	CG may either generate assembly code directly, or indirectly by generating retargetable CGR and PO.	-	If MS pursues instruction level parallelism (ILP) by means of pipelining or superscalar, a retargetable PO is necessary to optimize code to maintain the intended sequential semantics and take advantages of the ILP.
Instruction simulator (ISIM)	–	depends on the instruction set; however, most ISS's do not generate ISIM.			-	RTL-level ISIM is necessary to simulate the execution of assembly code on the synthesized processor.

Table 2.1 The categories of design automation for instruction set processors and their design objects and metrics. Approaches that span more than one category such as the work in this dissertation are not listed in this table.

*. NOTATIONS:
++: primary input specification or constraint to the design automation system
+: secondary input specification or constraint to the design automation system
--: primary output of the design automation system
-: secondary output, or side-effect of the primary output of the design automation system
DSP: digital signal processor
ASIP: application-specific instruction set processor
ISP: (general purpose) instruction set processor

the design of a retargetable code generator (code mapper) and a peephole optimizer (post-end optimizer).

2.2.3 Microarchitecture Synthesis (MS)

Given an instruction set specification, the microarchitecture synthesis approach synthesizes a microarchitecture at the register transfer level (RTL), usually including both data path and control path, which implements the given instruction set.

2.3 Automatic Instruction Set Design

Most of the early work on automatic instruction set designs view the design problem as a design process independent to the hardware implementation. Instructions were not restricted to single-cycle instructions since multi-cycle instructions can be supported through micro-programming (firmware). Without knowing the decode/control complexity, the focus was mainly in directly supporting high level languages or increasing the code density. The results were CISC-like instructions. These studies include Haney's [29], Bose's [4] and Bennett's [6] work. These techniques are not suitable for designing instruction sets for modern pipelined processors.

Sato et al. [72] proposed an integrated design framework for application specific instruction set processors. This framework generates profiling information from a given set of application benchmarks and their expected data. Based on the profiles, the design system customizes an instruction set from a super set (supported by the GCC compiler), and decides the hardware architecture (derived from the GCC's abstract machine model). Since this approach targets at the GCC environment, it has the advantage of having all the software development tools handy, e.g., the instruction simulator. All other instruction set synthesis tools have the disadvantages of the lack of instruction simulators for the instruction sets they generate. However, the instruction design space of Sato et al.'s approach is limited to the *a priori* GCC instruction set.

This framework is similar to our work in terms of the inputs and outputs of the design system; however, it is different from ours in terms of the machine model and the design method. They assume a sequential (non-pipelined) machine model, whereas we assume a pipelined machine with a data-stationary control model. On the other hand, Sato et al. generate instruction sets by customizing a super set, whereas we synthesize the instruction sets directly in order to find new and useful instructions for the given application domain.

Unlike previous approaches, Holmer [36] focused on generating instruction sets which closely couple to the underlying micro-architecture. As pipelined micro-architecture proved its superiority in 1980's, Holmer adopted the modern pipeline control model (data stationary control) and simple, parameterized data path as the underlying micro-architecture model. The parameters for a data path include the number of read/write register ports, memory ports, number of functional units and the cycle counts for memory operation. The user specifies the parameters, and then invokes the system to find the set of instructions which best utilizes the hardware resources such that minimal cycle counts for benchmarks are achieved. Our work builds on the results of Holmer by improving the problem formulation and synthesis algorithms. This makes it possible to conduct experiments with larger applications.

2.4 Microarchitecture Synthesis

The microarchitecture synthesis approaches the design problem in a different direction. The basic task of high level synthesis for pipelined instruction set processors is to construct pipelined micro-architectures at the RTL level, given any behavioral (abstract and sequential) specification of the instruction set architecture to be synthesized. It is assumed that the machine to be synthesized repeatedly executes some computation task.

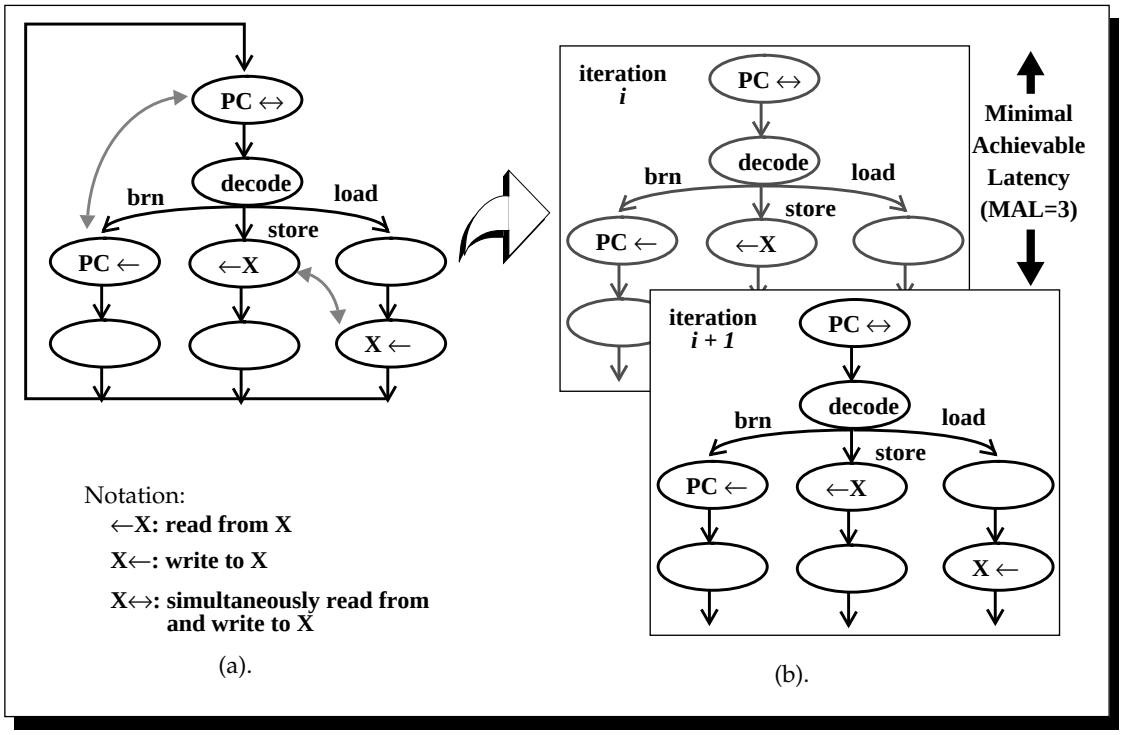


Figure 2.1 Pipeline synthesis and inter-iteration dependencies

2.4.1 The pipeline hazard problem in pipeline synthesis for instruction set processors

For instruction set processors, the computation task is the operations of fetch, decode and execution of instructions. The computation task is represented as a loop body in the specification, and the repetition is represented as the iteration of the loop body. Figure 2.1 (a) shows a state transition diagram of a three-instruction ISP: ovals are states with each state taking one clock cycle, thin arcs are state transitions, and the outer feed-back arc represents the iteration of the body. For simplicity we only show register accesses of *PC* and *X* in the states. The two dotted bi-directional arcs are inter-iteration dependencies[3] (inter-instruction dependencies) for registers *PC* and X, respectively.

3. The inter-instruction dependencies of instruction set processors appear as inter-iteration dependencies in the behavioral specification.

The task of pipeline synthesis is then to find an appropriate overlapping of multiple iterations of the loop body subject to the inter-iteration dependencies. As shown in Figure 2.1 (a), the longest inter-iteration dependency spans three cycles (the PC dependency); therefore, the optimal degree of overlapping without causing pipeline hazards is to issue an iteration every three cycles as shown in Figure 2.1 (b). The length of the longest inter-iteration dependency is called *minimal achievable latency* (MAL) [51], which has been used by several pipeline synthesis systems as the lower bound of the instruction initiation latency.

2.4.2 Synthesis approaches for instruction set processors

In the pursuit of high performance in digital circuit designs, many high level synthesis systems focus on automating the design of pipeline structures. Several pipeline scheduling algorithms have been developed. These algorithms differ in their application scope and achieved performance. SEHWA uses two algorithms: feasible scheduling and maximal scheduling [64]. HAL applies a modified force-directed scheduling algorithm to achieve better quality of pipeline scheduling and allocation at the cost of higher complexity [61]. These two systems do not work with applications which exhibit loop carried dependencies. PLS and CATHEDRAL II are capable of working on applications with LCD's by means of an iterative folding algorithm [46] and loop folding algorithm [27], respectively. Both systems adopt an iterative approach to find the MAL. This MAL then serves as the upper bound of the performance of the synthesized designs. The major application domain of the above synthesis systems is digital signal processors (DSP).

The System Architect's Workbench (SAW) synthesizes both un-pipelined and pipelined processors [71]. The possible biases of the designer's coding style in the instruction set specification are eliminated with a set of transformation rules to be selected by the user. The optimized specification is fed to a architectural partitioning

phase to determine the feasible hardware allocation. Finally the control step scheduling phase is invoked to assign operations to control steps. The System Architect's Workbench consists of two synthesis methodologies: one is tuned specifically to microprocessor designs, and the other supports a general design style. In the latter approach, pipelined designs are derived by interactively performing transformations at the system level. However, it does not attempt to solve pipeline hazards automatically.

There are several techniques available that try to minimize MAL by rearranging the micro-operations, in order to increase the degree of overlapping [15][46]. The limitation of these techniques is that they are still subject to the constraint of MAL. Pipelined instruction set processors with instruction initiation latencies less than MAL are impossible. For example, the MIPS R2000 [49] has MAL of two. If it were synthesized with these techniques, it would have instruction initiation latency of two, instead of one as in the manually designed one. Therefore, it can be concluded that these techniques are not suitable for synthesis of pipelined instruction set processors.

ASPD uses a VLIW-based architecture as the hardware template, and applies percolation scheduling and pipeline scheduling to explore the fine grain parallelism [7]. The system performs the design task in two phases: specification optimization and implementation optimization. In the first phase the above scheduling algorithms are used to generate the application micro-code. The VLIW template is customized in the second phase. ASPD is capable of synthesizing pipelined instruction set processors with an instruction initiation latency of one, regardless of their MALs. To resolve the induced pipeline hazards, the synthesized machine flushes the pipeline as long as an instruction which may cause hazards is decoded. Take the MIPS R2000 as an example, if it were synthesized by ASPD, the pipeline would flush as soon as a `delayed load`[4], `branch`

4. A delay load refers to the load instruction whose data to be loaded is not ready for the following instruction in the pipeline [32].

or `jump` is decoded, regardless whether the succeeding instructions are dependent instructions or not. This approach leaves no room for compilers to utilize the flushed (idle) cycles. In addition, the practicality of ASPD is limited by the difficulties of micro-code explosion and the complexity of pattern merging.

2.5 Code Generation

The problem of code generation is handled differently by researchers in the compiler community and digital signal processing (DSP) community. These are discussed in the following subsections.

2.5.1 Automatic generation of compiler backends

Compiler writers view code generation as two-step process: first, code generators and optimizers are constructed; second, the code generators and optimizers are used to compile applications into assembly code. To make the code generators and optimizers retargetable, compiler writers usually adopt modular approaches to separate machine-dependent information from machine-independent code generating and optimizing algorithms, as described in Section 2.1.3 on page 15. The machine-dependent information is usually kept in tabular formats. The machine-dependent information is derived from a machine description. Cattell's and Graham's works focus on the generation of code-generators [10][28], while Giegerich's work is on the derivation of machine-specific optimizers [26]. They derive their machine-dependent tables from instruction set semantics which describes the instruction formats, addressing modes, and operations. However, the micro-architectural features such as pipeline configuration are not considered in these systems. PIPER is different from these works in that it focuses on the construction of back-end compiler phases which are related to the micro-architecture. Currently, the

focus is restricted to generating the reorderers for instruction set processors with various pipeline configurations.

2.5.2 Code generation for DSP applications

Designers usually find that compilers provided by vendors of digital signal processors are not able to produce high performance code. The reason is that DSPs usually exhibit irregular data paths such as multiple small-size register files, many special registers, multi-port memory, multiple pipelined functional units, and chaining between functional units. Conventional retargetable code generators/optimizers do not perform well in these irregular data paths. Techniques in high level synthesis, such as scheduling and allocation, become attractive alternatives to code generation for these DSP cases.

2.6 Combined Approaches

The need to closely examine the problems of MS, ISS and ISM for instruction set processors has been noted by researchers, e.g., [36] and [66]. The most common way to solve the problems is the use of current tools with some iterative approaches.

In Kitabatake and Shirai's approach [50], the application is translated to an intermediate representation which is simulated. The operation frequencies and graph structure patterns are reported to the designer who then selects the desired patterns. The selected patterns define the instruction set and data path. The availability of a pattern may be affected by other selected patterns. However, due to the lack of reference, we are not sure how such interactions are handled in this approach.

Holmer and Pangrle propose the iteration between the instruction set design tool "Alchemy" and a high level synthesis tool "SandS" [38]. An initial data path model and its timing parameters are given to Alchemy which generates the initial instruction set from the model. The generated instruction set specification is then taken by SandS which

synthesizes both the detailed data path and control path, considering pipelining. Once the hardware details are obtained, the data path model for Alchemy is updated, and the design process repeats until the design objective is satisfied.

Praet, et al. uses a similar iterative scheme but the interaction is performed by the designer at a finer level [68]. This approach is intended for DSP processors with non-pipelined data path. The application is first analyzed for the types of operations and frequent common patterns. The initial data path is constructed by selecting modules from a cell library, based on the analysis results. The application is then mapped to the data path. The statistics of the mapped application are collected. A tool called *bundling* is invoked to find the possible chaining patterns in the application which are used to suggest the interconnection patterns for the data path. The interconnection patterns also define instructions. Another tool, *analyse*, is used to collect the usage patterns of the data path modules. Based on this information data path modules can be added or deleted. With such decisions, new interconnection patterns may be established. With the new data path, the application is mapped again, and the design process repeats until the design objective is satisfied.

Alomary et al.[1] cast the problem with a different flavor. The Harvard architecture is used as the hardware model. A super set of instructions that can be generated by the GNU C compiler is defined as the candidate set. The set of instructions are divided into three subsets: primary RTL (PRTL), basic RTL (BRTL), and extended RTL (XRTL). PRTLs are those that can be directly supported by the data path. BRTLs and XRTLs are complex ones which can be implemented with either PRTLs, microprograms or specialized hardware. The design problem is then expressed as selecting the best implementation methods for BRTLs and XRTLs such that performance/cost is optimized for the given application. A formal definition of the problem and its branch-and-bound solver are given in [2].

Like most of the related work, our approach synthesizes instructions from the given application, instead of customizing from a super set. However, our approach is different from the related work in that we are looking for a single formulation for the combined problem of instruction set design, microarchitecture design and code generation, instead of an iterative approach.

2.7 Comparison of ASIA, PIPER and Related Work

ASIA and PIPER can be compared to other work according to how they treat the design objects and metrics discussed in Section 2.1. The result is summarized in Table 2.2 on page 31. In the table, related work is grouped into categories according to Section 2.3 ~ Section 2.6. We use the same notations as in Table 2.2 on page 31. The '/ x' marks indicate that the corresponding design objects or metrics are irrelevant for some cases.

The comparison shows that ASIA has the same input/output specification as the related combined approaches in Section 2.6. The difference is the way the interaction between microarchitecture design and instruction set design is handled: ASIA uses an automatic and integrated scheme, while the related combined approaches use some manually controlled iterative schemes. On the other hand, PIPER is similar to related work in microarchitecture synthesis. However, the uniqueness of PIPER is the interface (the reordering table) to the compiler backend (peephole optimizer, or reorderer) which is generated during the pipeline synthesis process.

The table also shows the design objects and metrics that receive no or little attention in our and related approaches. This observation suggests some future directions. (1) Estimating cycle time at architectural and microarchitectural levels is a difficult task due to the lack of detailed information that is yet to be determined by design tools at the lower levels. (2) Estimation and minimization of power dissipation are important in

designing portable systems. Although there has been some work such as [75], further investigation is necessary to incorporate the power issue into synthesis. (3) Another missing link is the construction of the compiler backend: code generator and peephole optimizer. A new instruction set processor is of little use unless there is a compiler backend[5] for it. Although PIPER generates the reordering table for the reorderer, which can be part of the optimizer, there is still much more work to be done such as the pattern generation for peephole optimization and code generation. The construction of effective code generator and peephole optimizer is a tedious but nontrivial task. Automatic generation of retargetable code generators and peephole optimizers are thus desired. (4) Similarly, automatic generation of the instruction level simulators and microarchitectural level simulators are desired as well.

5. The compiler frontend, including lexical analysis, syntax analysis, semantic analysis and intermediate code generation, is machine-independent.

Design Objects & Metrics	Instruction Set Synthesis*	Code Generation	μ-architecture Synthesis	Combined Approaches		
	Haney[29], Bose[4], Bennett[5], PEAS [2], Holmer [36]	Cattell [10], Graham [28], Giegerich [26], C. & L. [13], L. M. & P. [56] S. T. & D. [75]	SEHWA[62], HAL[61], PLS[46], ASPD[7], etc.	K. & S. [50], H. & P. [38], Praet et al. [68], PEAS-I [1]	ASIA	PIPER
HARDWARE						
Architectural model	++	++	++	++	++	++
Control path			--			--
Data path	++/x	++	--	--	--	--
Chip area			-	-	-	-
Cycle count	--/x	--	-	--	--	--
Cycle time						
Power dissipation						
INTERFACE						
Instruction semantics	--	++	++	--	--	++
Instruction format	--	++	++	--	--	++
Instruction set size	--	++	++	--	--	++
SOFTWARE						
Application	++	++		++	++	
Assembly code		--	+/x	--	--	++
Code generator		–/x				
Peephole optimizer		–/x				--
Instruction simulator	-/x					

Table 2.2 Comparison between ASIA, PIPER and related work

*. NOTATIONS:

/x: The design object/metrics is irrelevant for some cases.

++: primary input specification or constraint to the design automation system

+: secondary input specification or constraint to the design automation system

--: primary output of the design automation system

-: secondary output, or side-effect of the primary output of the design automation system

Chapter 3 Design Framework

Section 3.1 describes the framework of the Advanced Design Automation System (ADAS), which is a full range design automation system for microprocessor design. Both ASIA and PIPER are part of the ADAS system. PIPER serves as the microarchitectural (behavioral) domain of ADAS, which takes an instruction set architecture specification as an input, and generates a reordering table as an interface to the compiler backend, and a pipelined microarchitecture at the register transfer level (RTL), including both data and control paths. ASIA, on top of PIPER, extends the capability of ADAS to the architectural level. ASIA reads in application benchmark programs, synthesizes an application-specific instruction set which can then be fed to PIPER as an input, and estimates the number of hardware resources to be allocated to the microarchitecture.

Following the description of ADAS, the inputs/outputs, major design phases and performance/cost estimation of ASIA and PIPER are summarized in Section 3.2 and Section 3.3, respectively.

3.1 The ADAS Design Automation System

3.1.1 Design levels of ADAS

ADAS accepts a set of application benchmarks and an architectural template as input, and produces as output a VLSI layout of a pipelined instruction set processor. In addition, ADAS also generates the application-specific instruction set, assembly code of the given application benchmarks and the reordering table which serves as the interface to the compiler backend ‘reorderer’. The new ADAS is an extension over the original ADAS [69], which was an improvement of the ASP design automation system [22].

ADAS spans many levels of design abstraction: architectural, microarchitectural (behavioral), logic and circuit, and physical (geometric) domains. Figure 3.1 shows the hierarchy of the new ADAS and corresponding design tools. The dashed line shows the design entry point of the original ADAS.

In the architectural domain, ASIA produces an application-specific instruction set and estimates the number of needed resources for the given architectural template

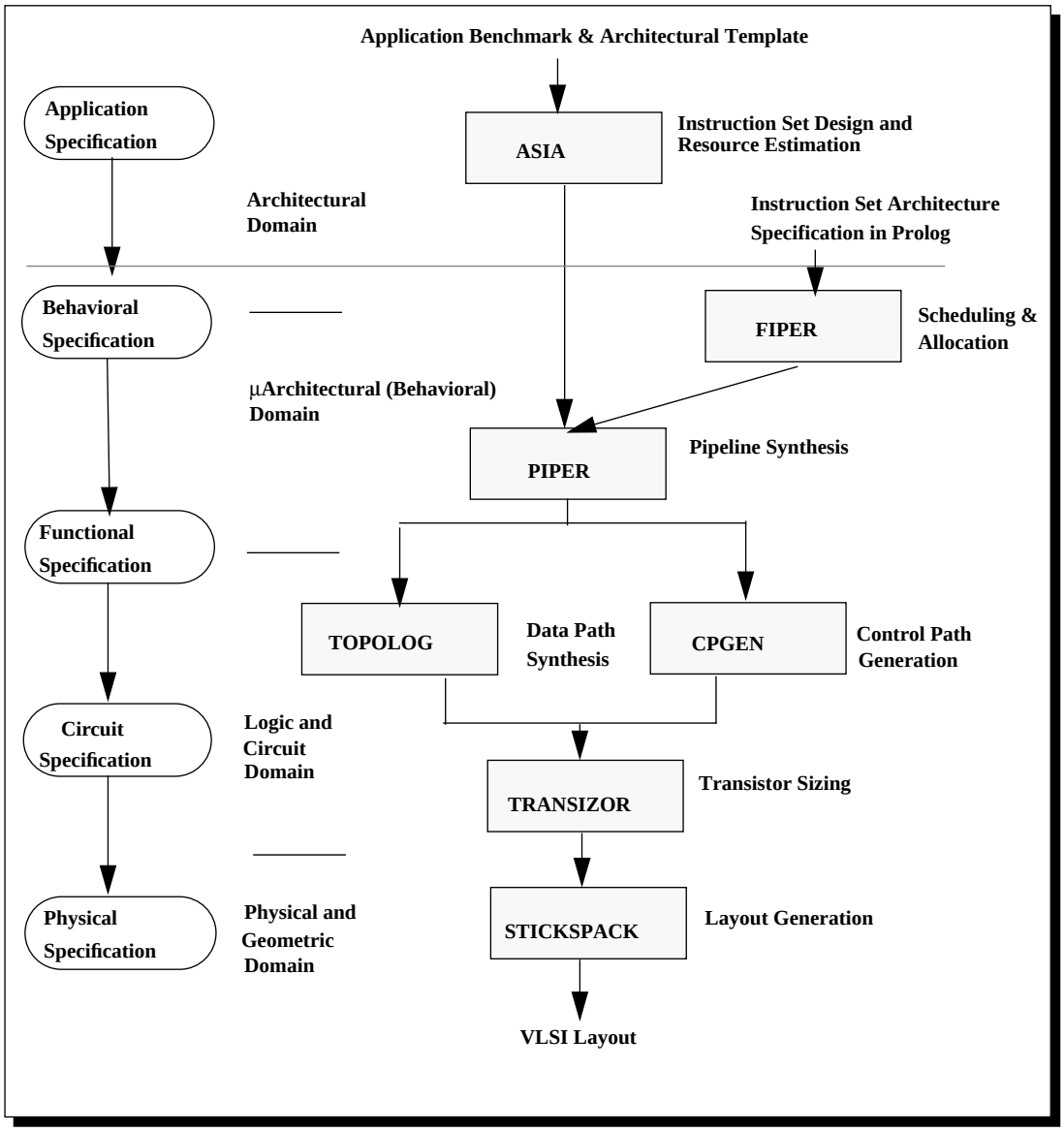


Figure 3.1 The new Advanced Design Automation System (ADAS)

and application benchmarks which represent the typical run-time environment for the processor to be designed. The synthesized instruction sets and the architectural template along with the allocated hardware resources are annotated as a finite state machine, which is passed down to the tool PIPER in the microarchitectural domain[1].

PIPER takes the finite state machine and transforms it into a pipelined machine. Both the data path and control path of the pipelined microarchitecture are generated at the RTL level. The data path is described in a net list of data path modules. The control path is described as a net list of control registers, glue logic and symbolic boolean equations for the control PLAs (programmable logic arrays). In addition, PIPER also generates a reordering table containing a set of reordering constraints which instruct the reorderer of the compiler backend how to properly reorder the sequence of the instruction stream to avoid pipeline hazards.

In the logic and circuit domain, TOPOLOG translates a data path description into a symbolic layout. The major tasks of TOPOLOG include cell allocation, cell placement and routing. CPGEN performs logic minimization and state assignment on the control path specification and creates a symbolic layout. TRANSIZER performs transistor sizing for the symbolic layouts of data path and control path to enhance circuit performance. TRANSIZER utilizes the control information generated by PIPER to avoid false paths when searching for critical paths in the circuit.

In the geometric domain, STICKPACK translates the symbolic layout into mask geometries in CIF format. The major tasks include compaction, placement and routing. A pad frame is also created automatically.

---

1. Another microarchitectural domain tool in the original ADAS is FIPER, but it is not used in the new ADAS. FIPER takes as input an instruction set architecture specification in Prolog, and transforms it into a finite state machine by scheduling and allocation processes. The output of FIPER is fed to PIPER for pipeline synthesis [69].

In ADAS, there is also a simulation pass which runs parallel with the synthesis pass. A set of application benchmarks are provided for simulation at major design domains to verify the correctness and performance of the design.

In the architectural domain, benchmarks are simulated at the instruction level to obtain frequencies of both individual instructions and consecutive instruction pairs. The frequencies are used by PIPER to estimate pipeline stalls due to pipeline hazards and to trade off hardware and software methods when resolving pipeline hazards in the microarchitecture.

In the microarchitectural domain, simulation is based on an event-driven finite state machine model. Clock ticks and interrupts represent events. The values of registers are compared with what are obtained at the architectural level cycle by cycle to verify design correctness.

In lower domains, simulation tools such as ESIM and IRSIM are used to produce traces of machine state for the synthesized processor at the circuit level. The machine code, translated from the application benchmarks, is used as the test vector for simulation. The produced traces are compared with what are produced by higher levels to ensure the design correctness. Finally, SPICE is used to estimate the cycle time of the synthesized processor.

3.1.2 Design philosophy

Unlike many traditional CAD tools[2], ADAS adopts an integrated and hierarchical approach to the design of microprocessors which spans many abstract levels: from architectural to VLSI mask level. A set of tools is carefully designed and implemented so

2. Most traditional CAD tools were developed independently. Although each individual tool is powerful and is applicable to a broad range of design styles, it is very difficult to integrate tools to accomplish design tasks which span many abstract levels. The major difficulty is the lack of appropriate interface languages and global understanding of related tools' functionality.

that they can be smoothly integrated with each other. The design tasks are carefully defined and partitioned into various tools, and interface languages between tools are well defined.

Figure 3.2 on page 36 illustrates the conceptual structure of ADAS. In ADAS, the design task is viewed as a sequence of design transformations. At each design stage, the design specification and constraints are successively transformed into equivalent, more detailed representation by a design generator. The design generator is controlled by design rules. The design rules include heuristics and modeling. In this design paradigm, design rules and design generators are separated, according to their roles in the design process: the design generators generate only correct designs, which are independent from the design rules that optimize over all possible correct designs. Designs can be

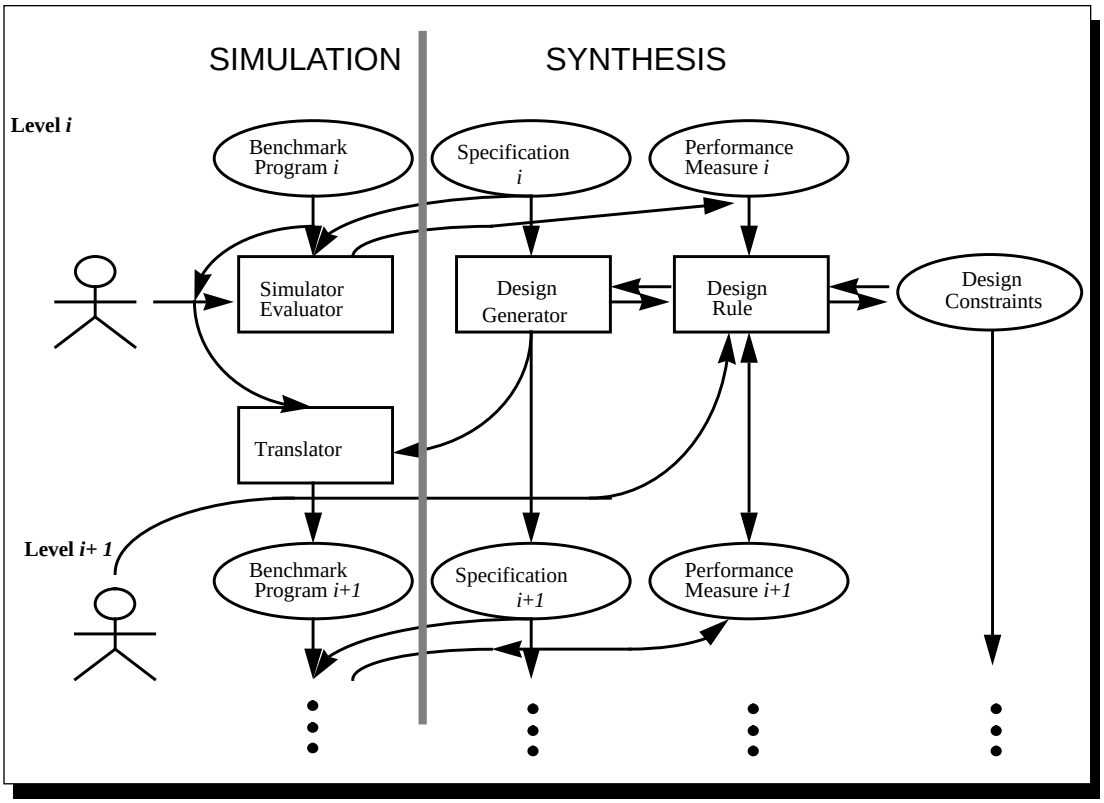


Figure 3.2 The conceptual structure of ADAS

improved by tuning the design rules. Also the experience from previous designs can be used to adjust the design rules. In addition to the design specification, the information from higher levels is carried down to lower levels to aid the design process whereever it is appropriate. For example, the register transfer patterns generated by PIPER in the microarchitectural domain are used by TRANSIZER in the circuit domain to eliminate false paths when looking for possible critical paths in the circuit.

Application benchmarks are provided to verify the design correctness and performance estimation. At each level there is a simulator for the benchmarks and a translator to map the benchmarks to the next level of detail. In addition, simulation results can be fed back to higher levels to improve the design rules.

3.2 Instruction Set Design and Resource Allocation: ASIA

Figure 3.3 depicts the overall structure of ASIA. The input/output, design flow and performance/cost estimation of ASIA are summarized in this section. Details of the design models, problem formulation and the algorithm are given in Chapter 4 and Chapter 5, respectively.

3.2.1 Input and output of ASIA

The input consists of the application benchmarks, an objective function, and a pipelined machine model. The application benchmarks are represented in terms of micro-operations that carry the semantic information of the programs. The micro-operations are not necessarily those of the target machine. The micro-operations can be derived from other architectures as long as these architectures have the same register configuration as those of the target machine. The objective function is a user-given function of the cycle count (representing the run time of the given benchmarks), the static code size, the instruction set size, and the hardware cost. The pipelined machine model

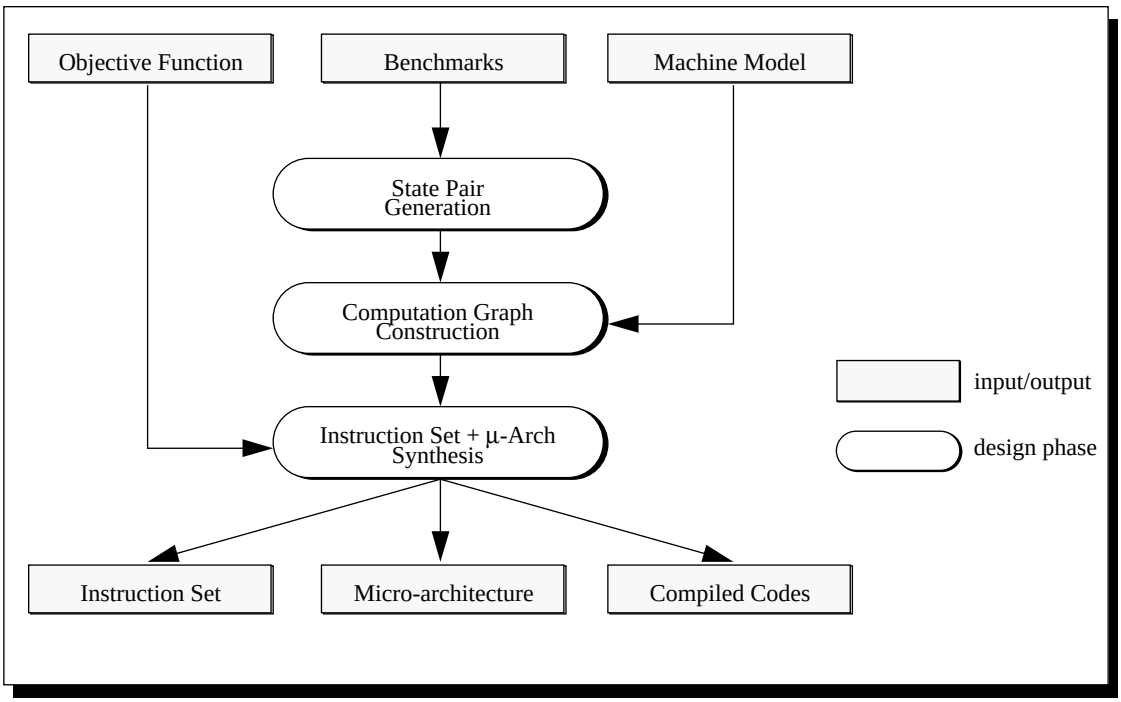


Figure 3.3 The ASIA system

allows for the specification of the pipeline stage configuration and the data path connection patterns.

The outputs consist of the instruction set and the hardware resources that optimize the given objective function, and also the compiled code which shows how the given benchmarks can be optimally compiled with the generated instruction set.

The models of the instruction set, the architectural template and the application benchmarks are given in Chapter 4.

3.2.2 Design process of ASIA

The design consists of three phases.

1. Transform the benchmarks into sequences of machine state transition

Each basic block of the benchmarks are transformed into a pair of machine state transition. The machine state is defined by the state of each architected (general/ special) register and memory. By representing the benchmarks as sequences of machine state transition, we are able to filter out the possible inefficiency embedded in the original micro-operations that describe the benchmarks. This approach also introduces a new view of the benchmarks, as sequences of machine states, instead of sequences of specific micro-operations that may unnecessarily over constrain the design space. Figure 3.4 shows a basic block and its corresponding initial and final states. Note that, for readability, the basic block is annotated as a sequence of instructions instead of micro-operations.

2. Generate computation graphs for these state pairs.

   The computation graphs are the data/control flow graphs (CDFG's) with the nodes being the micro-operations of the given machine model. For each state pair, the graph is generated by searching for the minimal number of micro-operations that move the initial state to the final state. Figure 3.6 shows a computation graph for the state pair in Figure 3.4.

   The machine model ASIA accepts a parameterized pipelined micro-architecture (Figure 3.5). The model allows us to explore different combinations of hardware features such as the connections between data path components, the cascades of pipeline stages, etc. Various computation graphs can be generated from the same state transitions by giving different micro-architecture models to the second phase of ASIA. Figure 3.7 depicts two variations of the micro-architectures. This approach frees us from being tied up to the micro-architecture from which the original micro-operations of the benchmarks are derived. We are able to target a wide range of micro-architectures.

3. Synthesize the instruction set and the micro-architecture from the computation graphs.

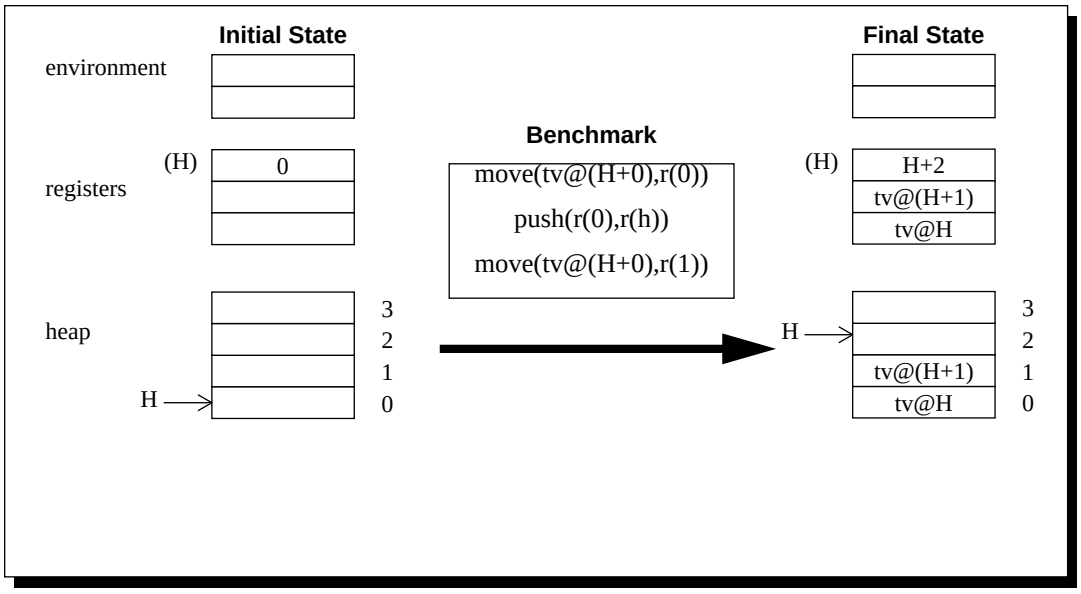


Figure 3.4 Machine state: an example [36]

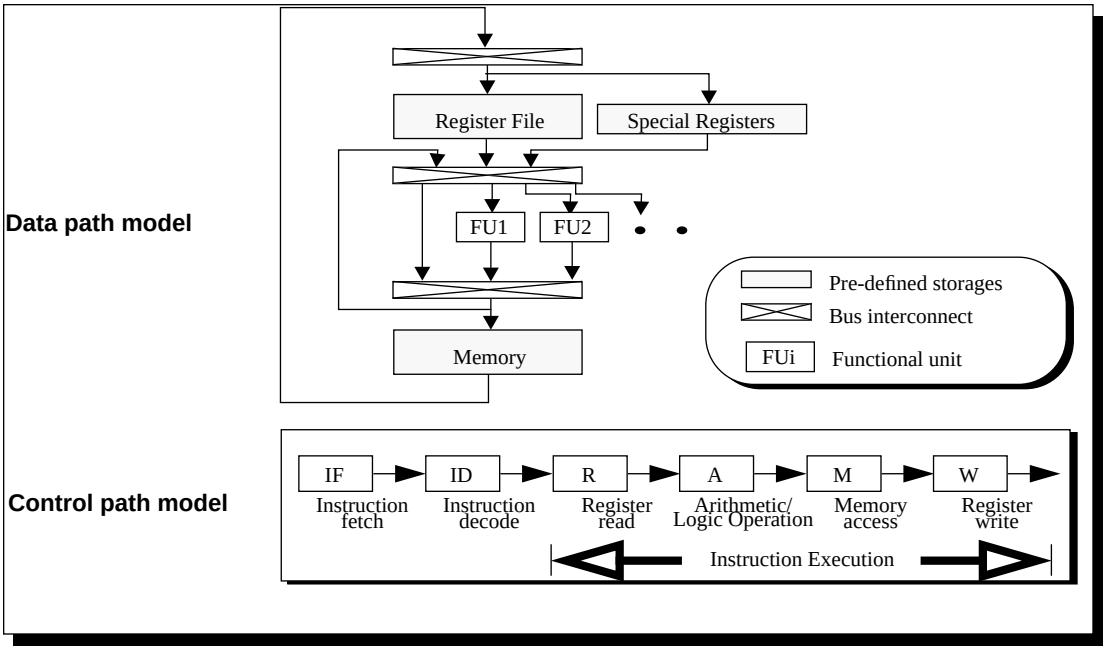


Figure 3.5 Parameterized micro-architecture model (data path) and pipeline control model (control path)

The designer specifies the bit widths of various instruction fields as well as the instruction word. The instructions are formed by packing the micro-operations in the computation graphs, subject to the constraints on the data/control dependencies and the instruction word width. While constructing the instructions, instruction fields and their instantiated values can be made implicit. For example, the instruction `inc(R1)` 'increment' is derived from the general form `add(R1,R1,1)`. The opcode `inc` implies that the source and destination are the same register, and the immediate data is always the constant of one.

The micro-architecture is generated by specifying the appropriate numbers of register read/write ports, ALU's, and memory ports.

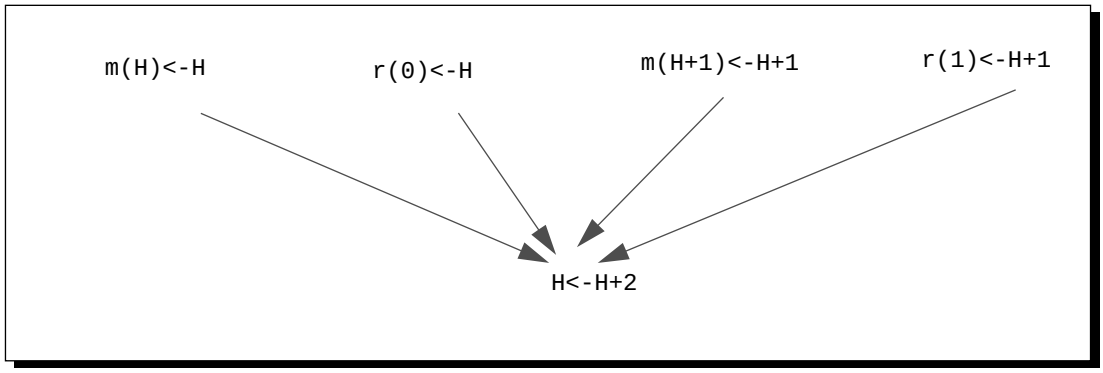


Figure 3.6 Computation graph: an example

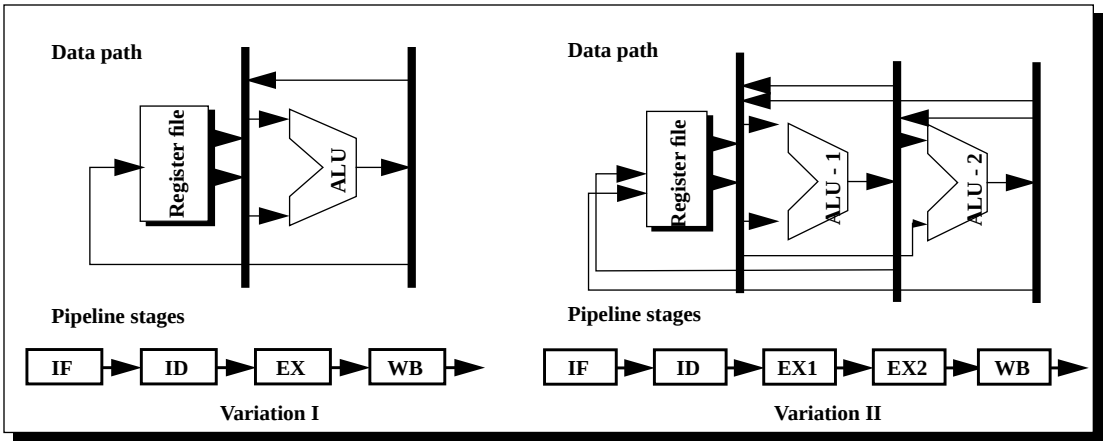


Figure 3.7 Variations of data paths and pipeline stages

The problem of concurrently synthesizing instruction sets and micro-architectures is formally stated, as opposed to what was empirically or ad hoc stated in previous approaches. The problem is mapped to the problem of simultaneous scheduling and allocation, which have been studied extensively in the area of behavioral synthesis. The idea is illustrated in Figure 3.8. The nodes in the computation graph are the objects to be scheduled. Each node consumes certain hardware resources and instruction fields. Each scheduled control step represents an instruction. The objective is to schedule the nodes so that the number of control steps (cycles) and resources used can be optimized, while the constraints on the instruction word width and the number of distinct control step patterns (instruction set size) are satisfied. In our approach, the problem of instruction field encoding is also addressed. By providing a clearer insight into the design problem, the formalization allows efficient exploration of the design space.

Figure 3.9 shows two possible designs of instruction sets and micro-architectures (Design I and II) for the computation graph in Figure 3.6. Design I has two instructions: ‘store and remember’ and ‘add’, which compile the graph into three cycles. The hardware resources needed by this design are 1R (register file read port), 1W (register file

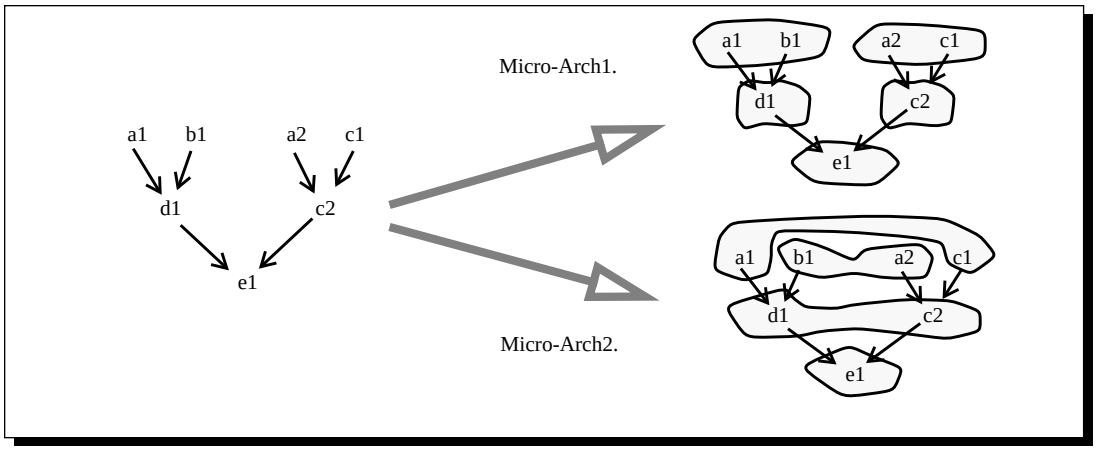
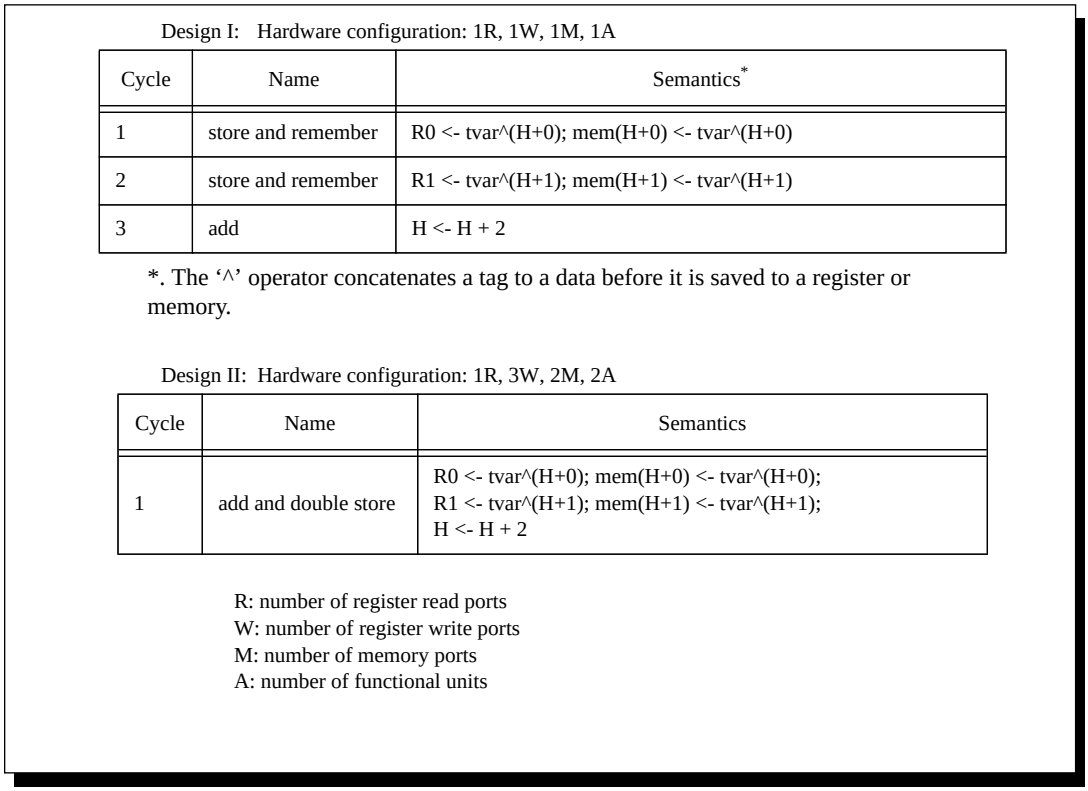


Figure 3.8 Mapping the design problem of instruction sets and micro-architectures as the simultaneous scheduling and allocation problem

write port), 1M (memory port) and 1A (ALU). Design II has only one instruction: ‘add and double store’, which compiles the graph into one cycle. This design has better performance than Design I, at the cost of more hardware resources: 1R, 3W, 2M, 2A. This example illustrates that there are many possible designs with different performance and cost tradeoffs for a given application benchmark. An objective function is thus required to balance the tradeoff between performance and cost.

The techniques used in step 1 and 2 are based on the idea of state-pairs described by Holmer in [36] and thus will not be further discussed in this dissertation. Interested readers may refer to his dissertation for more information. The techniques used in step 3 are the major focus in Chapter 4 and Chapter 5.



Design I: Hardware configuration: 1R, 1W, 1M, 1A

Cycle	Name	Semantics*
1	store and remember	R0 <- tvar^(H+0); mem(H+0) <- tvar^(H+0)
2	store and remember	R1 <- tvar^(H+1); mem(H+1) <- tvar^(H+1)
3	add	H <- H + 2

*. The ‘^’ operator concatenates a tag to a data before it is saved to a register or memory.

Design II: Hardware configuration: 1R, 3W, 2M, 2A

Cycle	Name	Semantics
1	add and double store	R0 <- tvar^(H+0); mem(H+0) <- tvar^(H+0); R1 <- tvar^(H+1); mem(H+1) <- tvar^(H+1); H <- H + 2

R: number of register read ports
W: number of register write ports
M: number of memory ports
A: number of functional units

Figure 3.9 Two possible designs of instruction sets and micro-architectures (Design I and Design II) for the computation graph (CDFG) in Figure 3.6 on page 41

3.2.3 Performance/cost estimation in ASIA

The cost of the synthesized instruction set is determined by the size of the instruction set and the numbers of register file read/write ports, memory ports and functional units. The global cost is calculated from these metrics. It is a function given by the designer.

The performance of the synthesized instruction set is measured in terms of the execution time of the given application benchmarks. The execution time is the product of the cycle count and cycle time. The cycle count is the number of clock cycles taken to execute the given application benchmarks, which reflects how well the instruction set matches the characteristics of the application benchmarks. The cycle time is the length of the clock, which is determined by the complexity of the control path, and the size and topology of the data path. It is difficult to estimate the cycle time at the architectural or microarchitectural level. A common approach is to use a lookup table which keeps records of the known cycle times for existing microarchitectures. The cycle times of new microarchitectures can be derived from the lookup table by using heuristics. In addition, the lookup table can be refined, based on detailed information fed back from lower design domains.

The objective function given by the designer is used to control the tradeoff between performance and cost. The objective function can be an arbitrary function of the metrics fore mentioned. An example of the objective function can be found in Section 5.3.2 on page 85.

The given objective function may inaccurately encapsulate the designer's idea. To solve this problem, ASIA's output can be run through PIPER and even lower domains of ADAS to obtain more implementation details. Such information can be fed back to modified ASIA's objective function and generate the instruction set and resource allocation again.

For example, suppose that the original objective function is `200ln(C)+I`. This function controls the tradeoff between the cycle count C and the instruction set size I, with which the designer tries to indirectly model the cost of the control path. However, the final control path synthesized by the lower domains of ADAS is about twice the expected size, indicating that the cost of the control path is underestimated by a factor of two in the original objective function. Therefore, the objective function can then be adjusted as `200ln(C)+2I` (or `100ln(C)+I`). As the number of variables in the objective function increases, many iterations of carefully controlled experiments are necessary in order to refine the objective function.

3.3 Synthesis of Microarchitectures and Compiler Backend Interfaces

3.3.1 Input and output of PIPER

Input

The inputs of PIPER consist of an abstract finite state machine specification, the benchmark statistics and a set of selected design options.

The abstract finite state machine specification represents an abstract, sequential register transfer implementation of the given instruction set architecture. The specification consists of a finite state graph and a description of architected registers. In the finite state graph, a state consists of a set of concurrent register transfers. The register transfers describe data movements between architected registers. The data movements may pass through functional units to perform some computation. The register transfers do not specify which interconnect the data goes through, how functional units are allocated, or how computation is assigned to functional units. These design tasks are to be completed by PIPER. A state transition is caused by a clock. A state may have many possible next states. In this case, the selection of the next state is controlled by some internal status of

the instruction set architecture such as the true/false signal, overflow/underflow signals, opcode decoding, etc.

The description of architected registers specifies the names of the registers which are visible to the instruction set, the register widths, and the definition of fields if the registers allow field-wise access[3].

The textual specification of the finite state graph for a 4-instruction (`add`, `load`, `store`, `brn`) processor SM0 is given in Figure 3.10 (a). The states are declared as Prolog predicates `state/3`. Each `state/3` predicate specifies one state: the first argument is the identifier for the state, the second argument is a list of concurrent register transfers occurred in the state, and the third argument specifies the next state. For example, in the state `bc(1)`, the register `memAR` gets the original value of the program counter `pc` and its value is incremented at the same time. At the next clock, the state goes to `bc(2)`.

Note that the next state can be multiple states, as in the case of state `bc(3)`, which is the decode state of the processor. There are four possible next states (`bc(5)`, `bc(12)`, `bc(14)`, `bc(18)`), based on the values in the `opcode` field of the register `memDR` (`add`, `store`, `load`, `brn`, respectively). A `switch/2` structure is used to specify the selection of next states. The first argument of the `switch/2` statement specifies the target on which the selection is based. The second argument is a list of cases and corresponding next states.

The predicates in Figure 3.10 (b) specify architected registers for the SM0 processor. The `element/5` predicates declare architected registers. The first argument specifies the type of the register and the second argument specifies the name of the register. For example, `ac` is a register of type `reg` (typical master-slaved register), whereas `pc` is a register of type `rsincreg` (resetable increment master-slaved register). The third

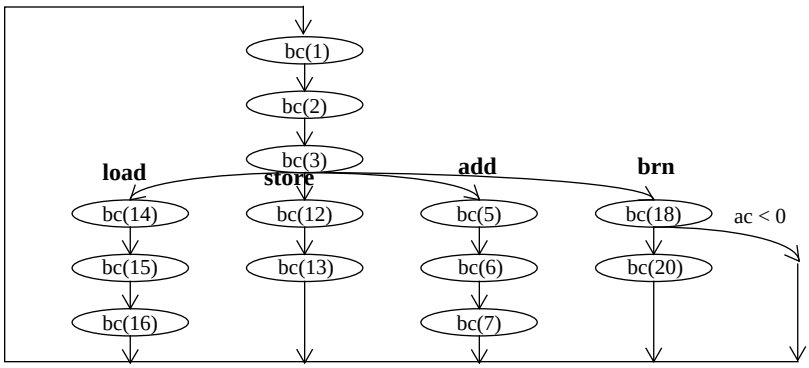
3. The field-wise access of a register refers to the capability of accessing the content of the register at the bit level. A group of bits can be defined as a field so they can be accessed as a group by referring to the field.

```
state(bc(1),[enable(pc,inc),move(pc,bus(b1),memAR),enable(memAR,load)],bc(2)).

state(bc(2),[enable(mem,mem_read)],bc(3)).

state(bc(3),[],switch(field(memDR,opcode),[case(add,bc(5)),case(store,bc(12)),case(load,bc(14)),case(brn,bc(18))])).

state(bc(5),[move(constant(binary(0),memAR:15-13),bus(b2),memAR),move(field(memDR,x),bus(b3), memAR),
enable(memAR,load)],bc(6)).

state(bc(6),[enable(mem,mem_read)],bc(7)).

state(bc(7),[move(ac,bus(b4),port(fu(fu1),1)),move(memDR,bus(b5),port(fu(fu1),2)),move(constant(0,port(fu(fu1),3)),
bus(b6),port(fu(fu1),3)),enable(fu(fu1),add),move(outport(fu(fu1),3),bus(b7),ac),enable(ac,load)],bc(1)).

state(bc(12),[move(ac,bus(b16),memDR),enable(memDR,load),move(constant(binary(0),memAR:15-
13),bus(b17),memAR),move(field(memDR,x),bus(b18),memAR),enable(memAR,load)],bc(13)).

state(bc(13),[enable(mem,mem_write)],bc(1)).

state(bc(14),[move(constant(binary(0),memAR:15-
13),bus(b19),memAR),move(field(memDR,x),bus(b20),memAR),enable(memAR,load)],bc(15)).

state(bc(15),[enable(mem,mem_read)],bc(16)).

state(bc(16),[move(memDR,bus(b21),ac),enable(ac,load)],bc(1)).

state(bc(18),[move(ac,bus(b24),port(fu(fu3),1)),move(constant(0,port(fu(fu3),2)),bus(b25),port(fu(fu3),2)),enable(fu(fu3),
encpr16)],switch(outport(fu(fu3),3),[case(false,bc(1)),case(true,bc(20))])).

state(bc(20),[move(constant(binary(0),pc:15-13),bus(b26),pc),move(field(memDR,x),bus(b27),pc),enable(pc,load)],bc(1)).
```

(a). Specification of the finite state graph

```
element(reg,ac,[],[],[]).
element(reg,memAR,[],[],[]).
element(reg,memDR,[],[],[]).
element(rsincreg,pc,[],[],[]).
stateRegister(ac,16).
stateRegister(memAR,16).
stateRegister(memDR,16).
stateRegister(pc,16).
stateField(memDR, x, 12-0).
stateField(memDR, opcode, 15-13).
```

(b). Specification of architected registers



(c). Graphical view

Figure 3.10 Finite state machine specification of a 4-instruction processor SM0

```
instruction_pattern_count([load,add],0.14).
instruction_pattern_count([store,add],0.13).
instruction_pattern_count([brn,load],0.01).
instruction_pattern_count([add,store],0.13).
instruction_pattern_count([brn,store],0.13).
```

Figure 3.11 Instruction pair frequencies

to the fifth arguments are not used. The `stateRegister/2` predicates specify the bitwidth of registers. All registers in SM0 are 16-bit wide. The `stateField/3` specifies the fields of the register. For SM0, two fields are defined for the register `memDR`: `opcode` and `x`, in the bit positions 15-13 and 12-0, respectively.

Figure 3.10 (c) shows the graphical view of the finite state graph. States `bc(1)`, `bc(2)` and `bc(3)`represent the instruction fetch and decode cycles. Four branches of `bc(3)`represent the execution of the four instructions respectively. For the branch instruction `brn`, there is a two-way branch following state `bc(18)` which checks the condition `ac<0`. At the end of the instruction execution, the next state goes back to state `bc(1)` to begin the fetch cycles for the next instruction.

Figure 3.11 shows an example of the frequencies of some consecutive instruction pairs obtained from the simulation trace of the given benchmark. This information is given to PIPER to estimate the run time performance of the synthesized pipelined microarchitecture and conduct the hardware/software trade-off.

Finally, Table 3.1 lists the available global synthesis options. *Instruction firing period* is the number of cycles between the firing of consecutive instructions. A maximally pipelined processor has the instruction firing period of one cycle. On the other hand, a nonpipelined processor has a fixed instruction firing period which is equal to the longest instruction execution latency or a variable instruction firing period which fires

the next instruction as soon as the current instruction finishes its execution. *Sequencer* refers to the sequencing style of the controller. The available choices include counters and finite state machines. Depending on the characteristics of the given instruction set architecture specification, each style has its advantages and disadvantages. It is up to the designer to perform several trial experiments to determine the best style for a given instruction set architecture. *State assignment* instructs PIPER to perform a simple state assignment for states and symbols, or generate an interface to invoke the Berkeley finite state machine compiler MEG to perform the assignment. The connection selects the bus style (using a combination of tristates and multiplexors) or the point-to-point style (using multiplexors exclusively) for connecting data path modules.

Option	Range
Instruction firing period	integer: 1 ~ longest instruction execution latency
Sequencer	counter or FSM
State assignment	ransom or assigned by the tool MEG
Connection	bus or point-to-point

Table 3.1 PIPER's synthesis options

Output

The output of PIPER consists of both hardware and software components. In the hardware, both the data path and the control path are generated. The data path is a net list of data path modules, including functional units, architected registers, internal registers, memory ports, I/O ports, I/O pads, tristates and multiplexors. The control path consists of boolean equations for control signals of each pipeline stage, glue logics for control signals from different pipeline stages, and the data/control path interface.

In the software, PIPER generates a reordering table which serves as the interface to the reorderer of the compiler backend. The reordering table contains a list of reordering constraints for dependent instruction pairs[4] which may have pipeline hazards when

both instructions of a specified pair are executed in some pipeline stages simultaneously. The reordering constraint specifies the number of instructions or NOP's (no-operation instructions) to be inserted between the instruction pair. The instructions or NOP's are inserted in order to maintain sufficient distance (*reorder distance*) between the pair so as to avoid pipeline hazards. Table 3.2 lists an example of the reordering table for a 6-stage pipeline implementation of the SM0 processor. For example, in the first data row is a dependent `load-brn` pair and its reorder distance of two.

Preceding instruction	Succeeding instruction	Reorder distance (number of instructions to be inserted)
load	brn	2
add	brn	2
brn	all instructions	4

Table 3.2 The reordering table for 6-stage SM0 processor

3.3.2 Design process of PIPER

PIPER maps the finite state machine representation of an instruction set architecture (ISA) into a pipelined register-transfer level design consisting of a data path and a control path. PIPER also generates an interface to the compiler back-end (reorderer), and measures its time and space complexities. The benchmark characteristics are measured to evaluate the quality of designs.

Figure 3.12 illustrates the conceptual structure of the PIPER system. PIPER takes as input the finite state machine specification and performs the following tasks: (1) pipeline assignment; (2) pipeline hazard resolution; (3) resource allocation.

The first phase, pipeline assignment, assigns micro-operations into pipeline stages. Pipeline hazards may be introduced by the pipeline assignment in highly pipe-

4. A dependent pair refers to a pair of instructions with the succeeding one depending on the results of the preceding one.

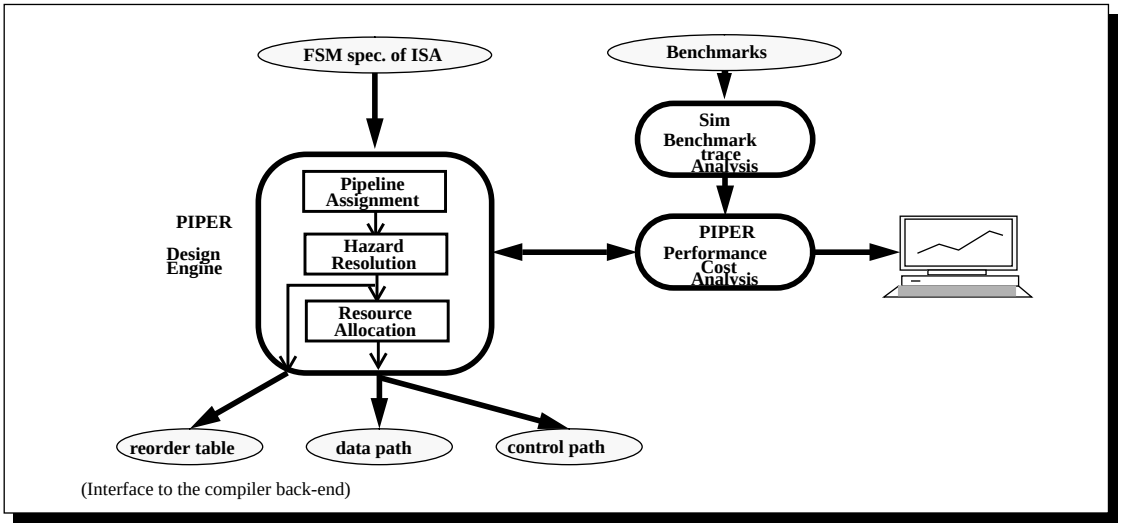


Figure 3.12 The structure of PIPER system

lined cases. These hazards are resolved in the second phase by a combination of hardware and software resolution strategies. This is accomplished in two conceptual steps: analysis of inter-instruction dependencies, and application of resolution strategies. In this phase PIPER generates a reordering table consisting of reordering constraints which instruct the compiler back-end (reorderer) to properly organize the codes for the pipelined machine synthesized. At the last phase, the hardware resources are allocated, producing a pipelined RTL level design.

Design decisions made by the pipeline scheduling and hazard resolution affect the performance and cost of hardware and the time/space complexities of the compiler back-end. Therefore, in addition to the design engine, there are a set of estimators and application benchmarks for the estimation of those effects.

The techniques used in the first and third phases are similar to other pipeline synthesis systems, and are not the focus of this dissertation. What distinguish PIPER from other systems is its second phase, which is presented in Chapter 4, Chapter 6 and Chapter 7.

3.3.3 Performance/cost estimation in PIPER

In this section an overview of performance and cost estimation is given. Detailed discussion is given in Section 7.2.

The cost of the synthesized microarchitecture is the estimated number of transistors. Data path and control path are estimated separately because of the significant difference between their structures. An empirical function is used to combine the two estimates into the chip cost. At present, routing area is not considered.

Not only hardware has a cost, but software has a cost as well. The cost of software is the time and memory used by the reorderer to compile code for the synthesized pipelined processor. The time/space complexity of the reorderer is usually a function of some characteristics of the reordering table such as the number of constraints and maximal reorder distance. Different pipeline implementations results in reordering tables with different characteristics, and therefore impose various degree of difficulty to the reorderer. Therefore, the cost of the reorderer can be obtained by substituting the characteristics of the reordering table into the time/space complexity functions.

In a hardware/software co-design environment, the hardware cost and software cost can be combined to construct a global system cost. The designer can specifies a global system cost function that is a function of hardware cost and software cost.

The performance of the pipelined processor includes two metrics: maximal speedup and average speedup of the given application benchmarks. The maximal speedup is a static measure which can be determined as soon as the pipeline structure is defined. However, performance is usually degraded from the maximal speedup due to pipeline stalls when executing real programs. The average speedup represents the actual speedup considering the run time pipeline stalls. The designer is required to provide PIPER an analytical model for estimating the average case.

It may happen that the cost/performance functions fore mentioned are not accurate. The output of PIPER can be fed to the lowers domains of ADAS for further details. The detailed information can be used to adjust the cost functions. On the other hand, assembly code of the given benchmarks can be reordered and simulated to obtain the run time performance. This information can be used to adjust the analytical model for the estimation of average speedup.

Chapter 4 Models: Hardware, Software and Interface

Modelling is an important way of understanding or describing complex systems. Models for the major design entities in ASIA and PIPER are presented in this chapter. The model of instruction sets is given in Section 4.1. The model of microarchitectures is given in Section 4.2. The model of application benchmarks is given in Section 4.3. The model of the interfaces for a compiler backend is given in Section 4.4.

4.1 Instruction Sets

An instruction contains one or more parallel micro-operations (MOPs). MOPs are controlled via instruction fields. The fields belong to some field types. For example, the instruction `add(R1,R2,Immed)` consists of an opcode field `add`, two register index fields `R1`and `R2`, and one immediate data field `Immed`. The width of the instruction word and field types is specified by the designer. Table 4.1 lists the specification of some instruction field types and their bit widths, taken from the BAM instruction set [8]. Each instruction has one opcode field, but the use of other fields is constrained only by the total number of bits needed by the operations in the instruction.

Instruction Field Type	Number of bits
instruction word	32
opcode	6
register (R)	5
tag (T)	5
displacement (D)	16
immediate (I)	16
relation (<,=,>,≠) operator (OP)	2

Table 4.1 Bit width specification for some instruction field types

Figure 4.1 lists the instruction formats for the instructions `add(R1,R2,Immed)` 'R_1<-R_2+Immed' and `inc(R)` 'R<-R+1', based on the bit width specification in Table 4.1. Note that there are 21 bits unused in the format of `inc`.

The operands of instructions can be encoded to become part of the opcode. There are two ways to encode operands. First, a specific value can be permanently assigned to an operand and becomes *implicit* to the opcode. Second, the register specifiers can be *unified*. For example, the instruction `inc` is obtained from the general instruction `add`. The facts of R_1=R_2 (unifying register specifiers; i.e., both register accesses refer to the same physical register) and Immed=1 (fixing an operand to a specific value which becomes implicit) are encoded into the opcode `inc`. Encoding operands saves instruction fields, at the cost of possibly larger instruction set size, additional connections and hardwired constants in the data path. For example, adding the instruction `inc` to the instruction set increases the instruction set size by one, and adds a hardwired constant '1' and an additional multiplexer in the data path, as shown in Figure 4.2.

Furthermore, encoding allows more MOPs to be packed into a single instruction. For example, if we find it happens very often that the values of two independent registers are increased by one at the same time, we may then devise a new instruction `inc(R1,R2)`which performs the MOPs 'R_1<-R_1+1; R_2<-R_2+1' (';' represents concurrency). This instruction uses only 16 bits, as opposed to 58 bits used by its generalized form 'R_1<-R_2+$Immed_1$; R_3<-R_4+$Immed_2$' which does not meet the instruction word width constraint for 32-bit instructions.

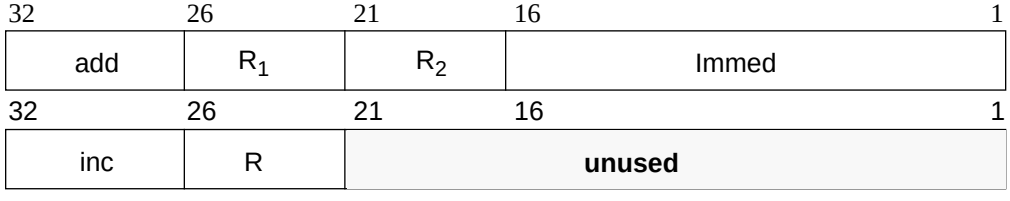
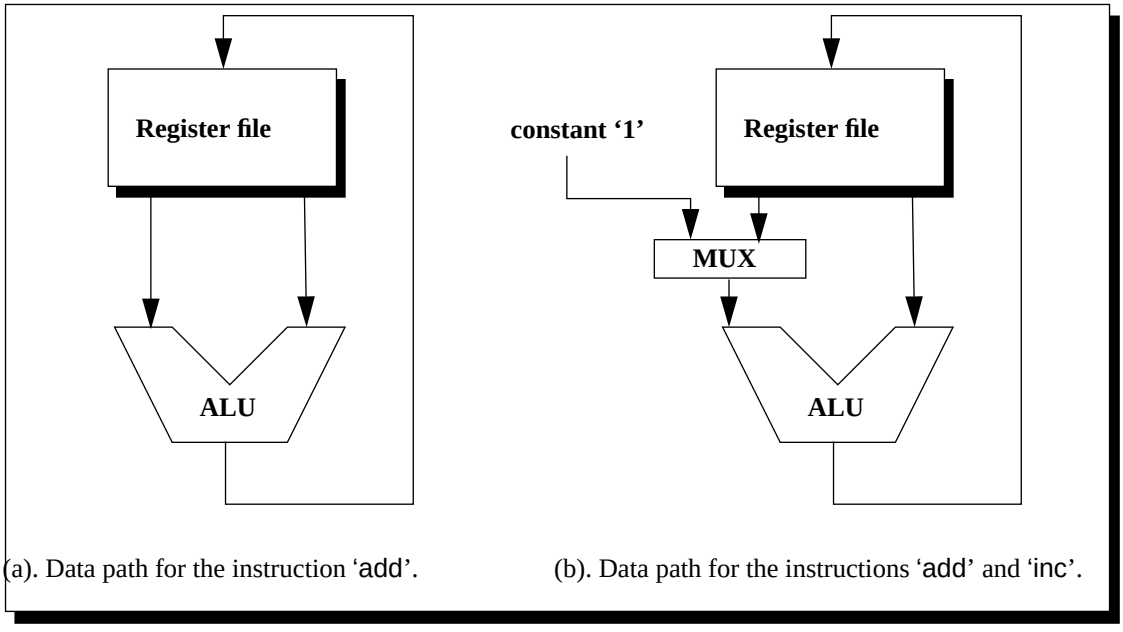


Figure 4.1 Examples of instruction formats

4.2 Microarchitectures

The design style of micro-architectures considered here is a linear, pipelined micro-architecture. The basic pipeline, as shown in Figure 4.3 (a), can be functionally partitioned into stages for instruction fetch (IF), instruction decode (ID), register read (R), arithmetic/logic operation (A), memory access (M), and register write (W). Each functional stage may take more than one cycle, and can be further pipelined. The first two stages are identical to all instructions. The last four stages, the *instruction execution stages*, are dependent on the semantics of the instructions. The pipeline stages can be varied. For example, the pipeline 'IF-ID/R-A-M-W', the case (b) in the figure, can be derived by merging the register-read stage with the instruction-decode stage, at the cost of restricting the instructions to single format for register specification such that registers can always be pre-fetched at the instruction-decode stage. On the other hand, the pipeline 'IF-ID-R-A/M-W', the case (c), is derived by merging the arithmetic stage with the memory stage, at the cost of eliminating the displacement addressing mode. The dis-



(a). Data path for the instruction 'add'. (b). Data path for the instructions 'add' and 'inc'.

Figure 4.2 Variation in data path for different instruction sets

placements have to be computed by other instructions which proceed the memory-related instructions.

The pipeline is controlled in a data stationary fashion [51]. In the data stationary control, the opcode flows through the pipeline in synchronization with the data being processed in the data path. Figure 4.4 shows the relationship between the control path with data stationary model and the data path. The register files at the top and bottom are the same register file. They are duplicated for the ease of readability. Opcodes are forwarded to each next stage synchronously. At each stage, the opcode, together with possi-

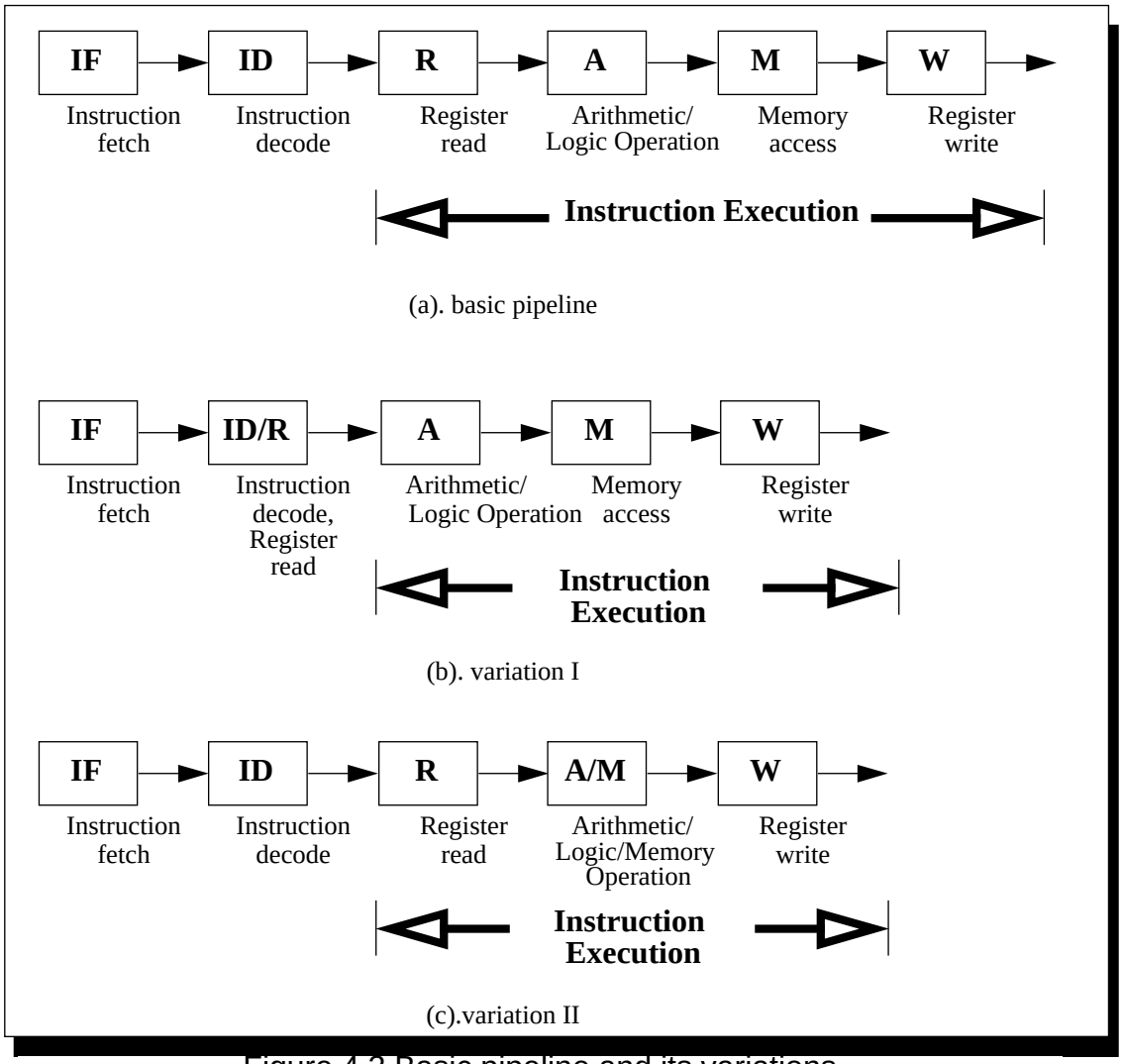


Figure 4.3 Basic pipeline and its variations

ble status bits from the data path, is decoded to generate the control signals necessary to drive the data path.

This pipeline configuration supports single-cycle instructions[1] which are typical of modern RISC-style processors. Multiple-cycle instructions can be accommodated with some modification to the linear pipeline such as the insertion of internal opcodes [67]. In order to manage the complexity of this research, general multiple-cycle instructions are not considered at this moment. However, multiple-cycle arithmetic/logic operations, memory access, and change of control flow (branch/jump/call) are supported by specifying the delay cycles as design parameters.

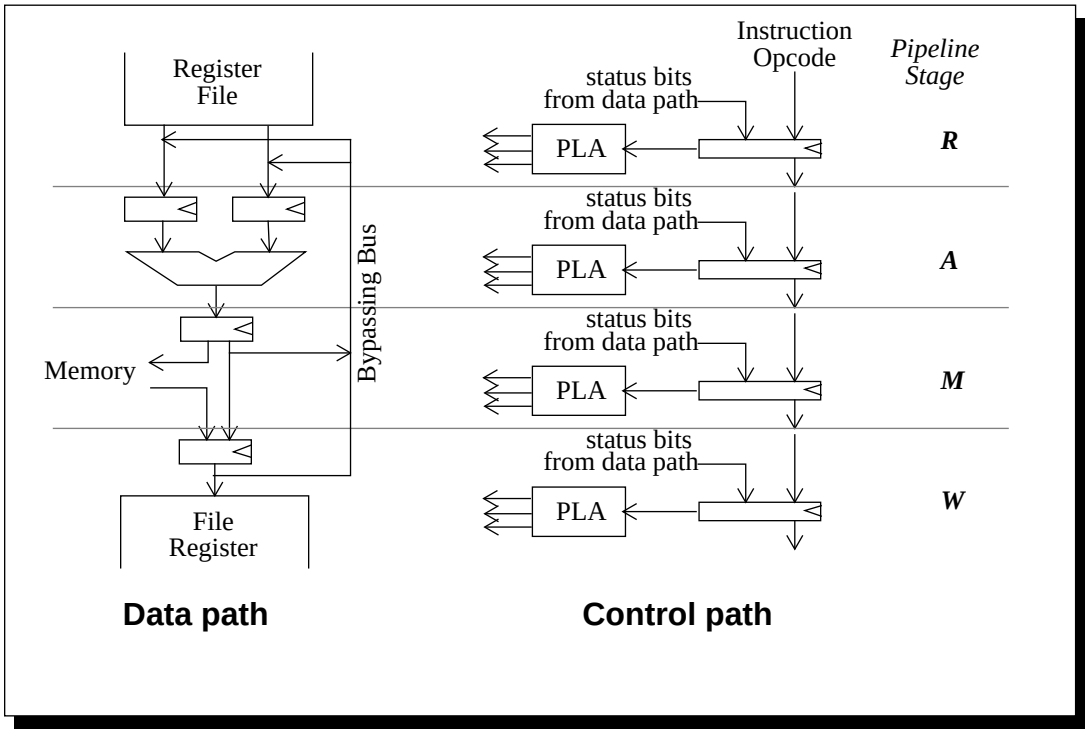


Figure 4.4 Data stationary control model

1. A *single-cycle* instruction has instruction latency of one cycle.

4.2.1 The specification for the target microarchitecture

The target microarchitecture can be fully described by specifying the supported MOPs and a set of parameters. The supported MOPs describe the functionality supported in the microarchitecture, and the connectivity among modules in the data path. For example, the first two columns of Table 4.2 list some of the MOPs supported in the VLSI-BAM microprocessor [35] and their corresponding MOP type IDs. The pipeline configuration 'IF-ID-R-A/M-W' in Figure 4.3 (c) can be derived by eliminating the MOPs `rmd`, `mrd` and `mrad` from Table 4.2.

Type ID	MOP*	Instruction Format Cost†	Hardware Cost‡
rr	R_1 <- R_2	R_1, R_2	1 R, 1 W
rra	R_1 <- R_1 + R_2	R_1, R_2	2 R, 1 W, 1 F
rrai	R_1 <- Immed + R_2	R_1, R_2, I	1 R, 1 W, 1 F
rrait	R_1 <- Tag^(Immed + R_2)	R_1, R_2, T, I	1 R, 1 W, 1 F
ri	R_1 <- Immed	R_1, I	1 W
rit	R_1 <-Tag^ Immed	R_1, T, I	1 W
rm	R_1 <- mem(R_2)	R_1, R_2	1 R, 1 W, 1 M
rmd	R_1 <- mem(R_2 + Immed)	R_1, R_2, I	1 R, 1 W, 1 M, 1 F
mr	mem(R_1) <- R_2	R_1, R_2	2 R, 1 M
mi	mem(R_1) <- Immed	R_1, I	1 R, 1 M
mrd	mem(R_1 + Disp) <- R_2	R_1, R_2, D	2 R, 1 M, 1 F
mrad	mem(R_1 + Disp) <- R_2+ Immed	R_1, R_2, D, I	2 R, 1 M, 2 F
jd	pc <- pc + Immed	I	1F

Table 4.2 MOP specification. Note that although not listed in the table, MOPs can be conditionally executed as well.

*. The operator '^' appends a tag to a value before the value is sent to a destination.
†. Refer to the notation in Table 4.1
‡. Notation: 'R'=read port of register-file, 'W'=write port of register-file, 'M'=memory port, 'F'=functional unit, and the value is the number of a particular hardware resource. For example, '2R' means two read ports for the register-file.

The set of parameters describes resource allocation and timing. The parameters include the number of register-file read/write ports, number of memory ports, number of functional units, the sizes of the register file and memory, latencies of operations, and the delay cycles between operations of memory access, functional units and control flow change. Table 4.3 is an example of the resource parameters for the VLSI-BAM microprocessor. The resource parameters specified in this table include the numbers and sizes of resources, and their operation latencies. Table 4.4 lists the delay parameters for various pairs of operations. For example, there should be one cycle delay between a memory operation and a succeeding (dependent) arithmetic operation, as specified by the M-A pair.

Resource Type	#	Operation	Latency
Read port; register file (R)	3	Register read	1
Write port: register file (W)	1	Register write	1
Memory read/write port (M)	2	Memory access	2
Functional unit (F)	1	Arithmetic/Logic	1
Register file size	32		
Memory size	2^{32}		

Table 4.3 Example of parameters for the target microarchitectures: resource size and latency

Operation pair	Delay cycles	Operation pair	Delay cycles
arithmetic-arithmetic (A-A)	0	memory-control (M-C)	1
arithmetic-memory (A-M)	0	control-arithmetic (C-A)	1
arithmetic-control (A-C)	0	control-memory (C-M)	1
memory-arithmetic (M-A)	1	control-control (C-C)	1
memory-memory (M-M)	1		

Table 4.4 Example of parameters for the target microarchitectures: delay cycles

Note that the existence of bypassing buses in the data path can be modeled by the delay parameters. For example, if we remove the bypassing bus in the ‘A’ stage in Figure 4.4, then the delay cycles for the A-A, A-M, and A-C pairs all become one, instead of zero.

Each MOP supported by the data path is assigned costs for the instruction format and hardware resources. The costs of the instruction format are the instruction fields required to operate the MOPs, including register index, function selectors, and immediate data. The hardware costs are the resources required to support the MOP. The hardware resources include read/write ports of the register file, memory ports, and functional units. The third and fourth columns in Table 4.2 lists the costs for the corresponding MOPs.

4.3 Application Benchmarks

Each application benchmark is represented as a group of weighted basic blocks. The weight is defined by the designers, and is usually used to indicate how many times the basic block is executed in the benchmark. The basic blocks are mapped to control/data flow graphs (CDFGs) of MOPs, based on the given MOP specification. Different microarchitectures result in different MOP specifications, which may map the basic blocks to different CDFGs. Figure 4.5 shows an example of a basic block, which consists of six MOPs, based on the MOP specification in Table 4.2. The bold labels before the MOPs are their IDs. The dashed arrows are control dependencies; the MOP `MO6` changes the control flow at the end of the basic block, and hence `MO6` logically follows MOPs `MO1~5`. The solid arrows are data-related dependencies. The data related dependencies can be characterized into three categories: *read-after-write* (RAW), *write-after-read* (WAR), and *write-after-write* (WAW). They all specify a *before* relation: the preceding MOP has to be scheduled before the succeeding MOP, except in micro-architec-

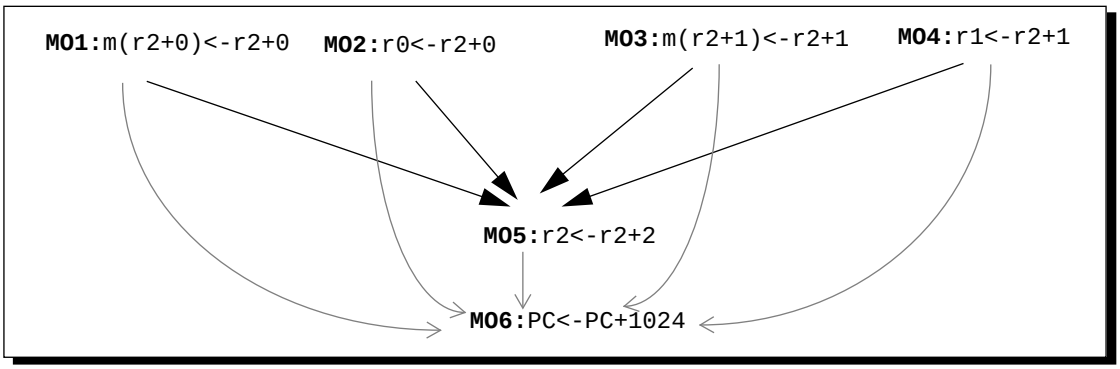


Figure 4.5 The control/data flow graph (CDFG) of MOPs of a simple basic

tures where master-slaved latches are used to implement registers. In this case, the WAR dependency indicates a *no-later-than* relation: the preceding MOP has to be scheduled no later than the succeeding MOP. The data dependencies in the figure are all WARs.

4.4 Interface to Compiler Backend

The design of compilers is closely related to the design of instruction set processors. In Figure 4.6 the design of compilers is partitioned into phases, according to the abstract levels of the corresponding hardware design. Figure 4.6 (a) shows three major abstract levels in the instruction set processor design. The design of an instruction set processor usually starts from defining its abstract machine model. The instruction set architecture is then defined as the realization of the abstract machine. At the next step, the microarchitecture (register-transfer level) is constructed to implement the instruction set architecture. Finally, the microarchitecture is further expanded into the VLSI layout (not shown in the figure). Most of microarchitectural (behavioral) synthesis systems deal with the design problems at the microarchitecture level.

In Figure 4.6 (b), the first level is the analysis phase including lexical, syntax, and semantic analyses which are not related to the machines. The second phase generates the intermediate code and may perform some optimization on the intermediate code. The intermediate code is usually derived from an abstract machine model which makes

the intermediate code generation *abstract machine dependent*. In the third phase, the intermediate code is expanded into the target machine code assuming that the instructions are sequentially[2] executed in the target machine. Some microarchitecture-independent optimizations may be performed here. The rules of expanding intermediate code to target machine code can be obtained when the semantics of the instruction set is defined; namely, when the instruction set architecture is defined. Therefore this phase of the compilation is *architecture dependent*. The major compilation tool in this phase is the code generator which performs the intermediate-code to target-machine-code mapping.

In the last phase of the compilation, the target machine code is modified to take advantage of the microarchitecture implementation and satisfy the constraints imposed

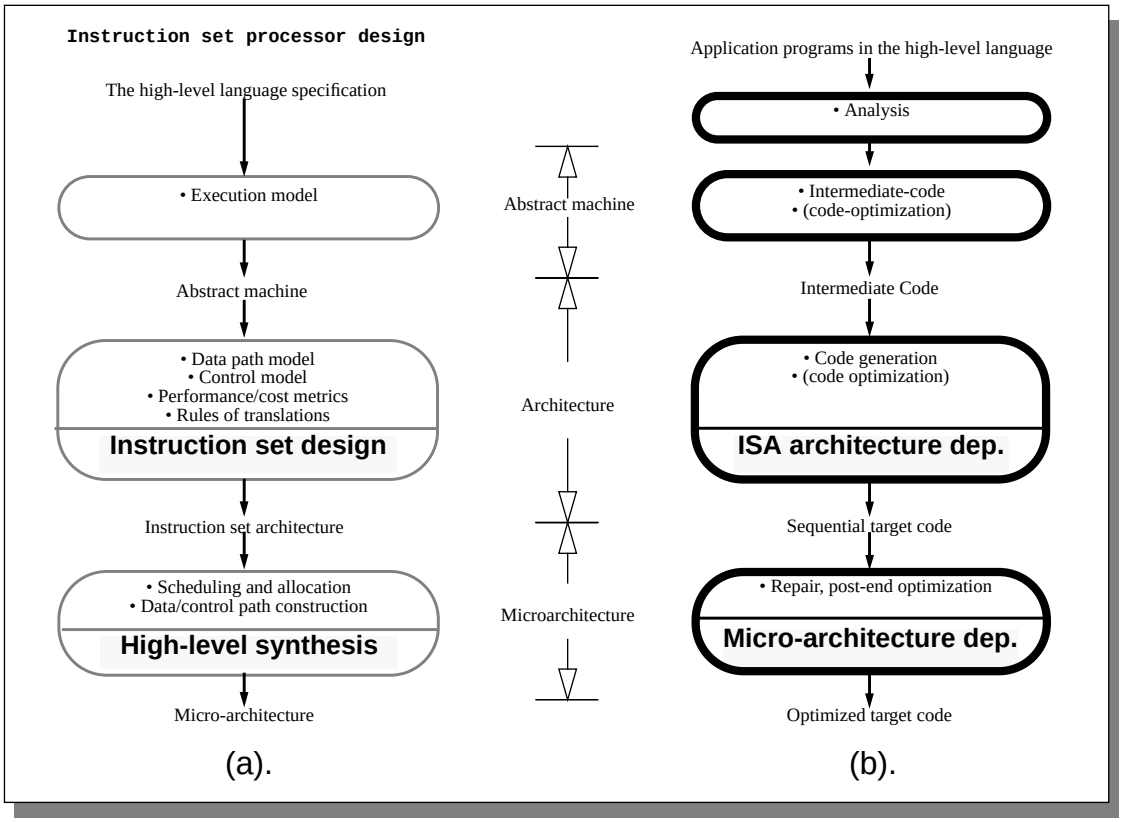


Figure 4.6 The phases of hardware and compiler design for instruction set processors

2. Pipelining or superscalar are not considered at this phase.

by such implementation. Typical considerations in this phase are insertion of *nop*'s, instruction reordering, and some optimization techniques which rely on the micro-architecture implementation. Therefore we term this phase as *microarchitecture dependent.* Typical compilation tools in this phase are the reorderer and peephole optimizer.

This dissertation adopts a concurrent engineering approach to the synthesis of hardware and compilation tools at the architectural and microarchitectural levels. Both hardware and software components of the corresponding levels are generated at the same time. It is assumed that the related compilation tools are retargetable; i.e., each tool consists of two modules: the algorithm body which is machine-independent and the machine information table which is machine-dependent. The retargetability is achieved by constructing the machine information table for each target machine. Based on this assumption, the concurrent engineering approach simultaneously generates both the hardware components and the machine information tables for the corresponding levels.

Due to time constraint, this dissertation only generates the machine information table (reordering table) for the reorderer at the moment. Generation of machine information tables for other compilation tools will be dealt with in future work.

The reordering table consists of reordering constraints for instruction pairs. Each constraint specifies how far a dependent[3] instruction pair, which may cause pipeline hazards in the pipeline, has to be separated. The distance (reorder distance) to be maintained between the instruction pair is measured by the number of independent instructions or `nop` instructions to be inserted between the pair. Table 3.2 on page 50 is the reordering table for a 6-stage pipeline implementation of the 4-instruction processor SM0 (shown in Figure 3.10 on page 47). The table is repeated here in Table 4.5 for easy reference. Each row in the table is a reordering constraint. The first and second columns specify the pre-

3. The dependence includes data, anti data, output and control dependency

ceding and succeeding instructions of a dependent instruction pair, respectively. The third column gives the corresponding reorder distance. Note that the instruction in the column of the preceding (succeeding) instruction can be a single instruction or a class of instructions. For example, there are three reordering constraints in the table. The first constraint specifies that the dependent instruction pair `load-brn`, i.e., a `load` instruction followed by a dependent `brn` (branch) instruction, has a reorder distance of 2. On the other hand, the third constraint has a succeeding instruction `all-instructions`, which means that a `brn` instruction followed by any dependent instruction[4] has a reorder distance of 4. This constraint effectively makes the `brn` instruction as a delay-branch instruction with four delay slots.

Preceding instruction	Succeeding instruction	Reorder distance (number of instructions to be inserted)
load	brn	2
add	brn	2
brn	all instructions	4

Table 4.5 The reordering table for a 6-stage SM0 processor

4. The next instruction of a branch instruction depends on the value of the program counter which is generated by the branch instruction. Therefore, there exist control dependencies between the destination instructions and the branch instruction.

Chapter 5 Synthesis of Instruction Sets and Microarchitectures

This chapter presents the problem formulation and algorithms for the synthesis of instruction sets and microarchitectures. First, the instruction set design is formulated as a modified scheduling problem in Section 5.1. A simulated annealing algorithm for the scheduling problem is presented in Section 5.2. The problem formulation and algorithm are extended to cover simultaneous synthesis of instruction sets and allocation of hardware resources for the microarchitectures in Section 5.3. The materials discussed in this chapter correspond to the third (synthesis) phase of the design process described in Section 3.2.2 on page 38.

5.1 Instruction Set Design as a Modified Scheduling Problem

The instruction set design problem can be formulated as a modified scheduling problem (Figure 5.1). The inputs of the problem are: an application represented in CDFGs, constraints of the instruction word and field widths and hardware resources, the objective function, and the microarchitecture specification. The MOPs in CDFGs are scheduled into time steps, subject to various constraints to be discussed later. While scheduling MOPs into time steps, instructions are formed at the same time. Finally, the outputs of this problem formulation is a synthesized instruction set and compiled code.

Two schedules of the MOPs in Figure 5.2 are shown in Table 5.1 and Table 5.2, respectively. In the first column of the table are time steps, and in the second column are the IDs of the MOPs scheduled into the corresponding time step. In this example we assumed a one-cycle delay for the jump `MO6` MOP and zero-cycle delay for memory operations. The schedule in Table 5.1 is a serialized one, with seven cycles. There is one

MOP in each time step. Note that there is a nop at the seventh cycle since MO6 is scheduled as the last MOP. The schedule in Table 5.2 is a more compact one, with four cycles. Note that the delay slot of MO6 is filled with MO5 so that there is no need for a nop.

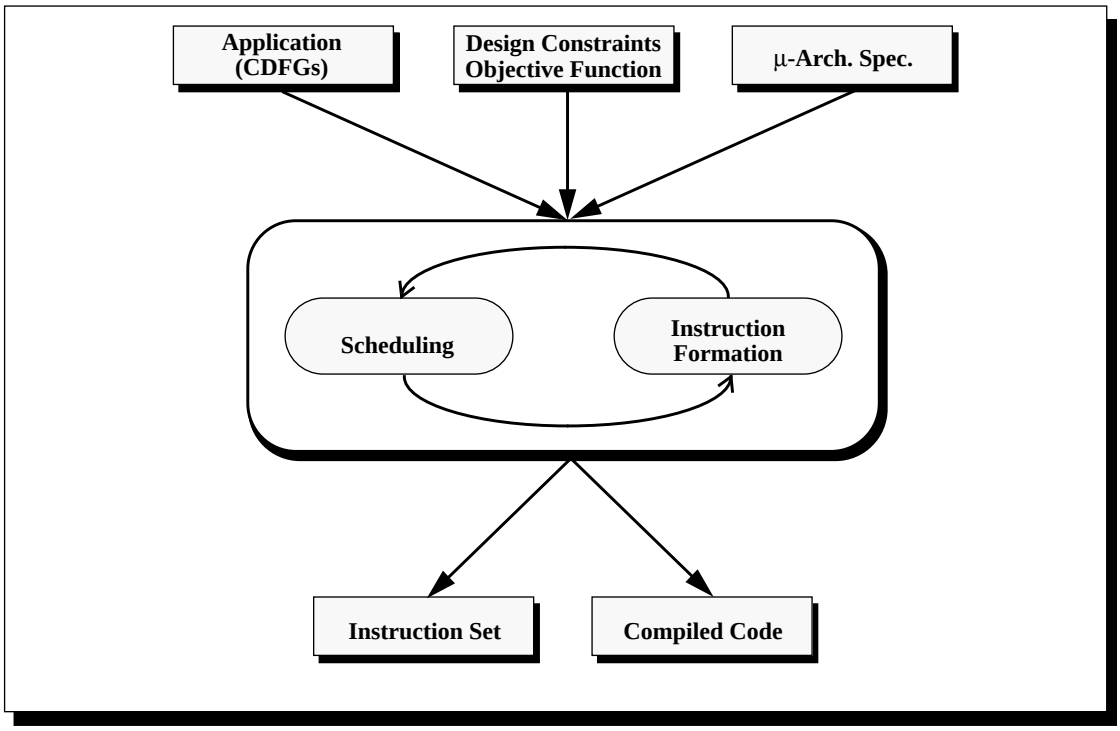


Figure 5.1 The integrated scheduling/instruction-formation process

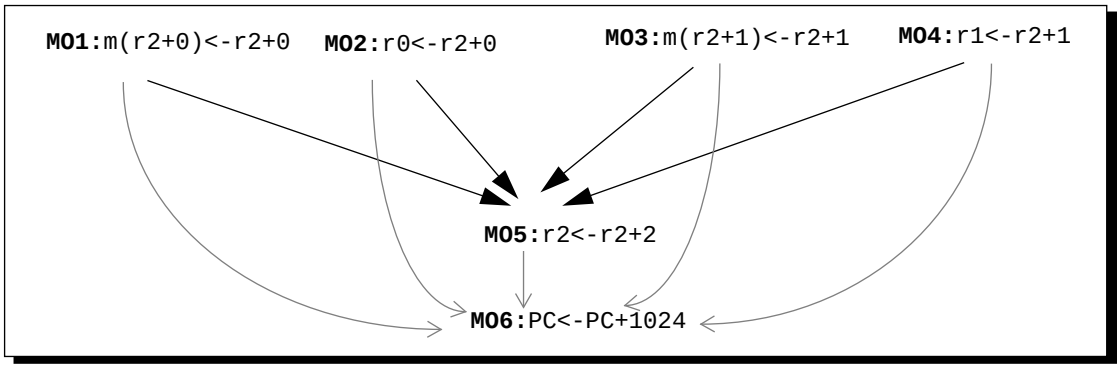


Figure 5.2 The control/data flow graph (CDFG) of MOPs of a simple basic

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time Step	mop IDs	RTLs	mop type IDs	Encoded fields	Inst. ID	Format	Field values	H/W cost*	Inst. word width
1	mo1	$m(R_1+D_1)<-R_2+D_2$	mrad		inst1	R_1, R_2, D_1, D_2	r2, r2, 0, 0	2R, 1M, 2F	48
2	mo2	$R_1<-R_2+I$	rrai	I=0	inst2	R_1, R_2	r0, r2	1R, 1W, 1F	16
3	mo3	$m(R_1+D_1)<-R_2+D_2$	mrad	$D_1=D_2$, $R_1=R_2$	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 7			Instruction set size 6			Hardware cost 2R, 1W, 1M, 2F		Max. instruction width = 68	

Table 5.1 Schedule I for the MOPs in Figure 5.2 and the resulted instructions

*. Notation: ‘R’=read port of register-file, ‘W’=write port of register-file, ‘M’=memory port, ‘F’=functional unit, and the value is the number of a particular hardware resource. For example, ‘2R’ means two read ports for register-file.

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time Step	mop IDs	RTLs	mop type IDs	Encoded fields	Inst. ID	Format	Field values	H/W cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$R_1=R_2=R_4$; $D_1=D_2=I$	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
2	mo3, mo4	$m(R_1+D_1)<-R_2+D_2$; $R_3<-R_4+I$	mrad, rrai	$R_1=R_2=R_4$; $D_1=D_2=I$	inst7	R_1, R_3, D_1	r2, r1, 1	1R, 1W, 1M, 1F	32
3	mo6	pc<-pc+D	jd		inst5	D	1024		22
4	mo5	$R_1<-R_2+I$	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
Cycle count 4			Instruction set size 3			Hardware cost 1R, 1W, 1M, 1F		Max. instruction width = 32	

Table 5.2 Schedule II for the MOPs in Figure 5.2 and the resulted instructions

5.1.1 Instruction formation: the binary tuple

The semantics of an instruction can be represented by a binary tuple <*MOPTypeIDs*, *IMPFields*>, where *MOPTypeIDs* is a list of type IDs (as shown in the first column of Table 4.2 on page 59) of MOPs contained in the instruction, and *IMPFields* is a list of fields that are encoded into the opcode.

For example, the binary tuple for the instruction `add(R1,R2,Immed)` is `<[rrai],[]>`. The instruction contains one MOP 'R_1<-R_2+Immed' with the type ID `rrai`, represented by the list in the first argument of the tuple. Since no fields are encoded, the second argument of the tuple is an empty list. On the other hand, the binary tuple for the instruction `inc(R)`, which is an encoded version of the instruction `add(R1,R2,Immed)` as discussed in Section 4.1 on page 54, is `<[rrai], [R1=R2,Immed=1]>`. The list in the second argument of the tuple specifies how the fields are encoded: The element `R1=R2` unifies the register specifiers R_1 and R_2 to the same register, and the element `Immed=1` fixes the immediate value permanently to the constant of one.

Instructions are generated from time steps in the schedule. Each time step corresponds to one instruction. The type IDs of the MOPs scheduled to the same time step are assigned to the first argument of the binary tuple for the instruction. The operand encoding specification, which is generated by an encoding process integrated into the scheduling process (described in Section 5.2), is assigned to the second argument of the binary tuple.

In Table 5.1 and Table 5.2, the columns under the header 'Instruction Semantics' and 'Instruction Fields' describe the semantics and field information of the instructions formed for the two schedules, respectively. The columns 'MOP type IDs' and 'Encoded fields' specify the binary tuples for the instructions. The RTLs for the corresponding MOP types are listed under the 'RTLs' column. Note that ';' denotes concurrency. The 'Inst Name' column assigns names to the generated instructions. The column 'Format'

describes the instruction format, i.e., the required instruction fields. The column ‘Field values’ lists the instantiated field values for the corresponding time step. Note that, in order to demonstrate the variation in the instruction formation, the instruction set in Table 5.1 is chosen from a non-optimal one.

For example, in Table 5.1, the MOPs scheduled into time step 4 and 5 have the same binary tuple, and thus are mapped to the same instruction `inst4(R1,R2,I)`, with their field values instantiated to `(r1,r2,1)` and `(r2,r2,2)`, respectively. Note that we use capitalized letters, e.g. `R1`, to denote the instruction fields, and non-capitalized letters, e.g. `r2`, to denote the instantiated values of the fields. On the other hand, the MOP in time step 2, is mapped to a different instruction `inst2(R1,R2)`, although it contains the same type of MOP `rrai` as in time steps 4 and 5. The reason is that its field for the immediate data `I` is permanently assigned to the constant ‘zero’ and made implicit in the opcode, as indicated by the specification `I=0` in the ‘Encoded field’ column. This implicit field makes the generated instruction behave as a ‘move’ instruction, instead of an ‘add’.

The compiled code can be obtained easily from the instruction names and instantiated field values. For example, the compiled code for the scheduled basic block in Table 5.2 is represented as the sequence: `inst7(r2,r0,0),inst7(r2,r1,1),inst5(1024),inst4(r2,r2,2)`.

The instruction set is formed by taking the union of instructions generated from all time steps. For example, the instruction set derived from the schedule in Table 5.1 contains six instructions (`inst1~inst6`), and the instruction set for the schedule in Table 5.2 contains three instructions (`inst4,inst5,inst7`).

5.1.2 Performance (cycle count) and Costs

The weighted sum of the lengths (number of time steps) of the scheduled basic blocks is the execution cycles of the benchmarks. The length of the basic block includes `nop` slots which are inserted by the design process to preserve the constraints due to multi-cycle operations. The design process will try to eliminate the `nop` slots by reordering other independent operations into the `nop` slots.

Each instruction has two costs associated with it. One is the total number of bits required to represent the instruction. This number is a summation of the opcode and explicit field widths that are required to operate the MOPs contained in the instruction. The implicit fields do not consume instruction bits. For example, in Table 5.1, the instruction `inst4` requires 32 bits, using the bit width specification in Table 4.1 on page 54; whereas `inst2` requires 16 bits only because its immediate data field is made implicit, saving 16 bits. The maximal bit widths of the instruction sets in Table 5.1 and Table 5.2 are 48 and 32 bits, respectively.

The second cost is hardware. It is the collection of the resources required by all MOPs contained in the instruction, minus the shared resources. The sharing of the resources can be related to field encoding. When two or more register reads of different MOPs are unified, i.e., reading from the same register, one read port of the register file is sufficient, instead of two or more. On the other hand, if more than one destination register receives results of the same arithmetic/logic expression, one functional unit is enough since the computation result can be shared. For example, `inst7` needs only one read port instead of three since `R1`, `R2`, and `R4` are unified. It also needs only one functional unit, instead of three, since the three destinations (memory data register, memory address register, register file) all receive the same value: `R1+D1`.

The global hardware resources are obtained by choosing the maximal number for each resource type from all instructions. For example, the global hardware resources

used for the schedule I and II in Table 5.1 and Table 5.2 are <2R, 1W, 1M, 2F> and <1R, 1W, 1M, 1F>, respectively.

The example in Table 5.2 shows that compact and powerful instructions can be synthesized by packing more MOPs into a single instruction, and making fields implicit and register ports unified to satisfy the cost constraints. This is particularly useful in an application specific environment where instruction sets can be customized to produce compact and efficient codes for the intended applications.

5.1.3 Constraints

The MOPs are scheduled into time steps, subject to several constraints. First, the data/control dependencies and the timing constraints (for multi-cycle MOPs) have to be satisfied. Data-dependent MOPs have to be scheduled into different time steps, subject to the precedent relationship and timing constraints, except single-cycle MOPs with WAR dependencies, which can be scheduled into the same time step if the registers can be read and written simultaneously. A control dependency with a timing constraint, e.g., a delayed jump, has to be dealt with differently. The MOPs that are data-independent to the jump/branch MOPs can be scheduled into the time steps before the jump/branch MOPs or the delay slots after the jump/branch MOPs. The length of the delay slots is determined by the timing constraint. For example, in Table 5.2, the independent MOP `MO5` is scheduled into time step 4, which is the delay slot of the jump `MO6`.

The second constraint is that the instruction word width and the hardware resources consumed by the instructions have to be no larger than what are specified by the designer. Third, the size of the instruction set has to be no more than what the opcode field can afford.

5.1.4 Objective function

Generally speaking, a richer instruction set may result in more compact and efficient compiled code. On the other hand, the larger the instruction set size, the more complex the decoding circuitry, and the more time the hardware designers spend in design and verification. The same trend holds true with respect to the compiler. Therefore, an objective function is necessary to control the performance/cost tradeoff.

The goal of our design system is to minimize the objective function. The objective function is a function of the cycle count *C* and instruction set size *S*, where *C* represents the performance metrics, or how many cycles the benchmarks execute on the target machine, and *S* represents the cost metrics. An interesting objective function suitable for our purpose is the following equation.

$$\text{Objective} = (100/P)\cdot\ln(C) + S \qquad \text{EQ 1}$$

This is an integral form, derived by Holmer in [36], of the statement “a new instruction will be accepted if it provides a P% performance improvement,” which tries to balance the instruction set size with the performance gain. Other types of objective functions can be used with the design system as well.

Note that in our formulation, the design constraints are checked separately, and are not captured in the objective function.

5.2 Simulated Annealing Algorithm

Although we have formulated the instruction set design problem as a scheduling problem, it is indeed more difficult than a regular scheduling problem. It is required to control the number of unique patterns (instruction set) in the time steps during the scheduling, in addition to the dependency and performance/cost constraints. Also, the problem

size is usually much larger than regular scheduling problems since the application benchmarks may easily contain thousands of MOPs which are to be scheduled.

We propose an efficient solution to the problem based on a simulated annealing scheme. An initial design state consisting of a schedule and its derived instruction set (generated by a pre-processor) is given to the design system, and then a simulated annealing process is invoked to modify the design state in order to optimize the objective function. The simulated annealing process is run until the design state achieves an equilibrium state.

Figure 5.3 lists the basic structure of our simulated annealing algorithm. In the outer `while` loop are the operations performed at each temperature point T. The temperature T is updated at the end of the operations. At each temperature, several movements (changes of the design state) are generated. The number of movements (M) generated is specified by the designer. How each movement is carried out is performed in the inner `while` loop.

```
/* Basic simulated annealing process */
1:  GIVEN: design state S, current temperature T, max. movement M;

2:  while (not achieving equilibrium state)
3:  {   C=0;
4:      while (C < M)
5:      {   if (violate constraints) Resolve_Constraint_Violation(S, S_next);
6:          else Generate_Next_State(S, S_next, T);

7:          if (Accept_Next_State(cost(S), cost(S_next), T)) then S= S_next;
8:          else S= S;

9:          C = C + 1;
10:     };
11:     T = Update(T);
12: }
```

Figure 5.3 The basic simulated annealing algorithm

In the following subsection, we present the move operators (Section 5.2.1) and heuristics (Section 5.2.3) for the procedures **Resolve_Constraint_Violation** and **Generate_Next_State**, the cooling schedule (Section 5.2.4) for **Update**, and the move acceptance rules (Section 5.2.5) in **Accept_Next_State**.

5.2.1 Move operators

The move operators are used to change the design state. They provide methods of manipulating the MOPs and time steps. The move operators can be characterized into three groups.

Manipulation of the instruction semantics and format

The first group manipulates the instruction semantics and format of a selected time step. There are five move operators in this group.

- *Generalization*: If the current instruction format of the selected time step contains encoded operands, make these operands general and become explicit in the instruction fields. The effects of this operator are increased instruction word width and hardware resources.
- *Unification*: Unify two register accesses in the MOPs; i.e., they always access the same register. For example, the specification of `R1=R2` in our previous example of the increment instruction `inc(R)` is a result of the ‘unification’ operator. The effects of this operator are the decreases in the instruction word width and/or register read/write ports.
- *Split*: Cancel the effect of the ‘unification’ operator. Two register accesses that were previously unified to the same register are made independent. The effects of this operator are the increases in the instruction word width and/or register read/write ports.

- *Implicit value*: Bind a register specifier to a specific register, or an immediate data field to a specific value. The specific values are the instantiated values in the MOPs of the selected time step. For example, the specification of `Immed=1` in the instruction `inc(R)` is a result of this operator. The effect of this operator is a decrease in the instruction word width.
- *Explicit value*: Cancel the effect of the 'implicit value' operator. Instruction fields that were previously bound to specific values are made explicit; i.e., their values are assigned by the compiler and are specified in the regular instruction fields. The effect of this operator is an increase in the instruction word width.

Manipulation of MOP's location

The second group of move operators involves the movement of the MOPs. There are four move operators in this group, which are all subject to the data/control dependencies and delay constraints when moving MOPs. The target MOPs and time steps can be selected randomly or with the guidance of heuristics.

- *Interchange*: Interchange the locations of two MOPs from different time steps. This operator changes the semantics and formats of the two instructions in the corresponding time steps.
- *Displacement*: Displace a MOP to another time step. This operator simplifies the semantics and format of the instruction in the original time step, and enriches the semantics and format of the instruction in the destination time step.
- *Insertion*: Insert an empty time step after or before the selected time step and move one MOP to the new time slot. This operator simplifies the semantics and formats of instructions in the selected and new time steps, and increases the cycle count.
- *Deletion*: Delete the selected time step if it is an empty one. This operator decreases the cycle count.

In our current implementation, if the selected MOPs contain unified or implicit fields, these fields are restored to the original forms (generalized, explicit) before the move operators in this group are applied to the MOPs. In addition to the mentioned effects, these move operators may change the resource usage in the selected time steps as well.

Microarchitecture-dependent operators

The third group of move operators includes methods that explore the special properties of the target microarchitecture. These move operators are provided by the designer as part of the microarchitecture specification.

For example, if the target microarchitecture provides both register file → functional unit → register file, and register file → register file data paths, then the designer can specify that the following MOPs (`rrai` and `rr`) are functionally equivalent and can be transformed from one to another:

`rrai`: `R1` ← `R2 + Immed (Immed=0)`

`rr`: `R1` ← `R2`

These MOPs have different costs in hardware and instruction format. While `rrai` uses a functional unit and consumes an additional instruction field for the immediate data, `rr` uses a direct bus between the read and write ports of the register file. When discovering an `rrai` MOP with its immediate data being zero, the design system can map this MOP to the equivalent `rr` MOP, or vice versa.

5.2.2 An example: changing the design state with move operators

We demonstrate how the move operators are used to change design states. Here we show a sequence of move operators which transforms the schedule and instruction set (one design state) in Table 5.1 to the ones (a better design state) in Table 5.2. The sequence is:

1. DISPLACEMENT: displace the `MO2` from time step 2 to 1 (as shown in Table 5.3).

2. UNIFICATION: unify fields D_1 and D_2 in the time step 1.

3. UNIFICATION: unify fields D_1 and I in the time step 1.

4. UNIFICATION: unify fields R_1 and R_2 in the time step 1.

5. UNIFICATION: unify fields R_1 and R_4 in the time step 1 (as shown inTable 5.4).

6. DELETION: delete the empty time step 2.

7. DISPLACEMENT: displace the `MO2` from time step 4 to 3 (as shown in Table 5.5).

8. DELETION: delete the empty time step 4.

9. UNIFICATION: unify fields D_1 and I in the time step 1.

10. UNIFICATION: unify fields R_1 and R_4 in the time step 1.

11. DISPLACEMENT: displace the `MO5` from time step 5 to 7 (as shown in Table 5.6)

12. DELETION: delete the empty time step 5.

Table 5.3 ~ Table 5.6 show the resulting schedule and instruction set for the design state after the first, fifth, seventh and eleventh move operators are applied, respectively. After the twelfth move operator is applied, the design state in Table 5.2 is obtained. In the last row of the tables we show the cycle count, instruction set size, hardware cost, and instruction word width for the corresponding design states. The deleted time steps are shown as shaded rows. The time steps in which the move operators are applied are emphasized with heavy rectangles around the time step indices. The ele-

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time Step	mop IDs	RTLs	mop type IDs	Encoded fields	Inst. ID	Format	Field values	H/W cost	Inst. word width
1	mo1, **mo2**	$m(R_1+D_1)$<-R_2+D_2; **R_3<-R_4+I**	mrad, **rrai**		inst11	R_1, R_2, **R_3, R_4**, D_1, D_2, **I**	r2, r2, **r0, r2**, 0, 0, **0**	**4R**, **1W**, 1M, **3F**	68
2									
3	mo3	$m(R_1+D_1)$<-R_2+D_2	mrad	$D_1=D_2$, $R_1=R_2$	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 7			Instruction set size 5			Hardware cost **4R**, 1W, 1M, **3F**		Max. instruction width = **68**	

Table 5.3 The design state after the application of the first move operator

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time Step	mop IDs	RTLs	mop type IDs	Encoded fields	Inst. ID	Format	Field values	H/W cost	Inst. word width
1	mo1, mo2	$m(R_1+D_1)$<-R_2+D_2; R_3<-R_4+I	mrad, rrai	$D_1=D_2$=I,R_1 =R_2=R_4	inst7	R_1, R_3, D_1	r2, r0, 0	**1R**, 1W, 1M, **1F**	32
2									
3	mo3	$m(R_1+D_1)$<-R_2+D_2	mrad	$D_1=D_2$, $R_1=R_2$	inst3	R_1, D_1	r2, 1	1R, 1M, 1F	27
4	mo4	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r1, r2, 1	1R, 1W, 1F	32
5	mo5	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 7			Instruction set size 5			Hardware cost **1R**, 1W, 1M, **1F**		Max. instruction width = **32**	

Table 5.4 The design state after the application of the fifth move operator

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time Step	mop IDs	RTLs	mop type IDs	Encoded fields	Inst. ID	Format	Field values	H/W cost	Inst. word width
1	mo1, mo2	m(R_1+D_1)<-R_2+D_2; R_3<-R_4+I	mrad, rrai	D_1=D_2=I,R_1 =R_2=R_4	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
2									
3	mo3, mo4	m(R_1+D_1)<-R_2+D_2; **R_3<-R_4+I**	mrad, rrai	D_1=D_2, R_1=R_2	inst31	R_1, **R_3, R_4**, D_1, **I**	r2, **r1, r2,** 1, **1**	**2R**, **1W**, 1M, **2F**	48
4									
5	mo5	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	nop		nop		inst6				6
Cycle count 6			Instruction set size 5			Hardware cost **2R**, 1W, 1M, **2F**		Max. instruction width = **48**	

Table 5.5 The design state after the application of the seventh move operator

Schedule		Instruction Semantics				Instruction Fields		Costs	
Time Step	mop IDs	RTLs	mop type IDs	Encoded fields	Inst. ID	Format	Field values	H/W cost	Inst. word width
1	mo1, mo2	m(R_1+D_1)<-R_2+D_2; R_3<-R_4+I	mrad, rrai	D_1=D_2=I,R_1 =R_2=R_4	inst7	R_1, R_3, D_1	r2, r0, 0	1R, 1W, 1M, 1F	32
2									
3	mo3, mo4	m(R_1+D_1)<-R_2+D_2; R_3<-R_4+I	mrad, rrai	D_1=D_2**=I**, R_1=R_2**=R_4**	inst7	R_1, R_3, D_1	r2, r1, 1	**1R**, 1W, 1M, **1F**	32
4									
5									
6	mo6	pc<-pc+D	jd		inst5	D	1024		22
7	mo5	R_1<-R_2+I	rrai		inst4	R_1, R_2, I	r2, r2, 2	1R, 1W, 1F	32
Cycle count 5			Instruction set size 3			Hardware cost **1R**, 1W, 1M, **1F**		Max. instruction width = **32**	

Table 5.6 The design state after the application of the eleventh move operator

ments in the design state that are modified by the move operators are listed with bold characters. Note that, for ease of illustration, we use the original time step indices in Table 5.1 in the above sequence when referring to selected time steps. In the implementation, the indices of time steps have to be adjusted when time steps are inserted or deleted such that the delay constraints between MOPs can be correctly maintained.

Note that there are more than one sequence which accomplish the same design state transition. How such sequences are formed depends on the design algorithm. In our simulated annealing scheme, the move operators are selected with a mix of random and heuristics strategies as described in Section 5.2.3.

5.2.3 Heuristics for target selection

During each iteration, the design space is examined to determine whether it violates the design constraints. If it does, a time step is randomly selected from a pool of time steps with violating constraints. If more than one constraint is violated, the resource constraint gets higher priority than the instruction word width constraint since a movement that resolves the former may resolve the latter as well.

Depending on the type of the constraints, one of the following rules is applied.

1. If the instruction word width constraint is violated, apply randomly one of the move operators: ‘unification’, ‘implicit value’, ‘interchange’, ‘displacement’ or ‘insertion’;

2. If the resource constraint is violated, apply randomly one of the move operators: ‘unification’ (only when the register port constraint is violated), ‘implicit value’, ‘displacement’ or ‘insertion’.

When the current design space does not violate any constraint, all move operators are eligible for changing the design state. In this case, a basic block is selected with

the probability $Selection_i$, which is the selection weight of a basic block i and is defined by the equation shown below. F_i is the execution frequency of the basic block i in the benchmark, N_i is the number of MOPs in the basic block i, and the summation in the denominator is the total number of MOPs executed in the benchmark. Therefore, the selection weight is intended to denote the degree of importance of a basic block in the benchmark. A time step is then randomly chosen from the selected basic block, and one move operator is randomly selected and applied to the time step.

$$Selection_i = \frac{F_i \cdot N_i}{\sum_i F_i \cdot N_i}$$

5.2.4 Temperature and cooling schedule

A proper initial temperature should accept virtually any transition (see page 154 of [55]). Several trial runs may be required to set the initial temperature. A simple heuristic to select the initial temperature is to start with a given temperature and see if all transitions are accepted in this temperature. If not, double the value of the initial temperature and repeat the trial until all transitions are accepted.

The number of movements tried at each temperature (M) is proportional to the total number of MOPs in the benchmarks, typically five times, which is controlled by the designer. The next temperature is 90% of the current temperature. A low temperature point is defined such that special handling routines can be applied to stabilize the design state. The annealing process terminates when the design state stays unchanged for a certain (e.g., four) consecutive temperature points.

5.2.5 Move acceptance

At high temperatures, a movement that satisfies one of the following conditions is definitely accepted.

1. The movement reduces the value of the objective function;

2. The movement is a result of constraint resolution; i.e., it is a necessary movement in order to resolve some constraint violations.

Otherwise, a movement is accepted with the probability of $exp^{-(\Delta/T)}$ where Δ is the increased value of the objective function and T is the current temperature.

At low temperatures, a different strategy is adopted to stabilize the design state. A movement is accepted when either one of the following conditions is true.

1. The movement generates a new state which does not violate any design constraint and has a lower objective value;

2. The movement is a result of constraint resolution. This condition is the same as the one at high temperatures.

Otherwise, only those movements that generate new states which do not violate any design constraint are accepted with the probability of $exp^{-(\Delta/T)}$.

In addition, the current best design state is kept when the algorithm decides to accept inferior design states. If the design state reached by the last accepted movement in each temperature point is inferior to the current best state, the design state falls back to the current best state with the probability $1-T/T_i$ where T_i is the initial temperature. This action is adopted because we observed in our early experiments with the prototype [44] of the algorithm that the simulated annealing process sometimes was not able to reach an equivalent or better state after it jumped out of a local best state.

5.3 Extension for Microarchitecture Design

The previous formulation can be extended to synthesize the microarchitecture at the same time, with the introduction of the architecture template specification language,

resource allocation, a proper objective function, and design constraints which are discussed in this section.

5.3.1 Resource allocation

Hardware resources are allocated while MOPs are scheduled to time steps. This is similar to the problem formulation in [23]. For each time step, the required hardware resources are the total of the resources consumed by each MOP scheduled into the time step, minus the resources that are shared. The sharing of resources in a time step is due to the operand encoding. When two or more register reads belonging to different MOPs are unified, i.e., reading from the same register, one register read port is sufficient. Also, if more than one destination register receive results of the same arithmetic/logic expression, one functional unit is enough since the computation result can be shared.

For example, the instruction `add_store(R1,R2,R3,R4,Immed)` 'R_1<-R_2+Immed; m(R_3)<-R_4' (';' represents parallelism) requires *three* register read ports for the register read specifiers `R2`, `R3` and `R4`, one register write port for `R1`, one functional unit for addition, and one memory port for the store operation. On the other hand, the instruction `push(R1,R2,Immed)`'R_1<-R_2+Immed; m(R_2)<-R_2' requires only *one* register read port, as opposed to three, with the requirements for other types of resources remaining unchanged. The saving is due to the unification of register read specifiers R_2=R_3=R_4.

The global resources allocated is then the union of the resources used by each instruction.

5.3.2 Design constraints and objective function

The design constraints used in Section 5.1.3 on page 72 remain intact, except the resource ones which are eliminated from the problem formulation. The algorithm is responsible for finding the best resource allocation according to the objective function.

The goal of the algorithm is to minimize the objective function which is an arbitrary function of execution time *T*, static code size *S*, instruction set size *I*, and hardware cost *H*. An example of the objective function is EQ 2 where *f*, *g*, *h* and *k* are sub-functions which calculates the contributions of its corresponding parameters. The sub-functions are given in EQ 3, EQ 4, EQ 5 and EQ 6, where C_1 through C_7 are constants defined by the designer.

$$Objective = f(T) + g(I) + h(H) + k(S) \quad \text{EQ 2}$$

$$f(T) = C_1 \bullet ln(T) \quad \text{EQ 3}$$

$$g(I) = C_2 \bullet I \quad \text{EQ 4}$$

$$h(H) = C_3 \bullet (\text{\# of register-file read port}) + C_4 \bullet (\text{\# of register-file write ports}) + C_5 \bullet (\text{\# of memory ports}) + C_6 \bullet (\text{\# of functional units}) \quad \text{EQ 5}$$

$$k(S) = C_7 \bullet ln(S) \quad \text{EQ 6}$$

The natural logarithmic form of EQ 3 (EQ 6) is suggested by Holmer in [36] as a way to balance the tradeoff between the instruction set size and the execution time (static code size). The objective function EQ 2 indicates that the following design changes are favorable:

1. increasing the size of the instruction set by one results in the decrease of cycle count by at least $(100C_2/C_1)\%$ or the decrease of static code size by at least $(100C_2/C_7)\%$, assuming other parameters are constant;

2. increasing the number of the register-file read ports by one results in the decrease of cycle count by at least ($100C_3/C_1$)% or the decrease of static code size by at least ($100C_3/C_7$)%, assuming other parameters are constant;

3. increasing the number of the register-file write ports by one results in the decrease of cycle count by at least ($100C_4/C_1$)% or the decrease of static code size by at least ($100C_4/C_7$)%, assuming other parameters are constant;

4. increasing the number of the memory ports by one results in the decrease of cycle count by at least ($100C_5/C_1$)% or the decrease of static code size by at least ($100C_5/C_7$)%, assuming other parameters are constant;

5. increasing the number of the functional units (ALU's) by one results in the decrease of cycle count by at least ($100C_6/C_1$)% or the decrease of static code size by at least ($100C_6/C_7$)%, assuming other parameters are constant.

Other forms of more sophisticated objective functions can be used as well.

5.3.3 Design process

The design process for co-synthesis of instruction sets and microarchitectures consists of three phases.

1. The given application is translated to dependency graphs of MOPs which are supported by the given architecture template. This translation is performed in two steps. First, the application, written in a high level language, is translated into an intermediate representation by the compiler of the high level language (in our current environment, the Aquarius Prolog Compiler [79]). Second, a retargetable

MOP mapper, consulting the given architectural template specified with the language described in Section 4.2.1 on page 59, transforms the intermediate representation into the dependency graphs of MOPs.

2. A preprocessor generates a simple-minded schedule for the MOPs. An instruction set is derived from the schedule. This is done by directly mapping time steps in the schedule into instructions without encoding any operand. The obtained schedule and instruction set constitute the initial design state.

3. The simulated annealing algorithm, with the modifications discussed in Section 5.3.1 and Section 5.3.2, is invoked to optimize the design state. Note that the initial temperature for the annealing process has to be higher than the problem in Section 5.2. A simple heuristic to set the initial temperature is to adjust the initial temperature such that there is no rejection of states with high costs at the initial temperature. The number of movements tried at each temperature has to be larger as well. These modifications are due to the much larger design space when instruction sets and microarchitectures are designed together. Several experiments may be necessary in order to set the proper values for these parameters.

The best instruction set, microarchitecture, and assembly code which minimize the objective function can be obtained after the design state reaches the equilibrium state.

Chapter 6 Taxonomy and Resolution for Inter-instruction Dependency

PIPER, which constructs pipelined microarchitectures and reordering tables, is the microarchitectural domain synthesis tool of ADAS. As described in Section 3.3.2 on page 50, the major synthesis phases of PIPER are (1) pipeline assignment; (2) pipeline hazard resolution; (3) resource allocation. What makes PIPER unique to other approaches is the hardware/software concurrent engineering approach in the second phase. The second phase can be divided into two tasks: analysis of inter-instruction dependencies and application of hardware/software resolutions, which are presented in Section 6.1 and Section 6.2, respectively. Techniques developed in these sections are for pipelined processors that fire one instruction every cycle. Section 6.3 discusses extension of the techniques in previous sections for synthesis of pipelined processors with instruction firing period of more than one cycle.

6.1 Extended Taxonomy for Inter-instruction Dependencies

Data dependent pipeline hazards are caused by inter-instruction dependencies. To resolve a hazard, we have to determine the type of dependency it involves and choose an appropriate resolution strategy. We first provide a classification of inter-instruction dependencies that consists of nine classes, derived from the cross products of <forward/backward/stationary> and <data/anti/output>.

The general model of the pipeline structures that we consider is a single pipeline, with linearly cascaded stages and constant stage latency[1]. This model is similar to the

1. The *stage latency* is the number of clock cycles that an instruction spends in one pipeline stage before it advances to the next stage.

pipelined SISD machine model defined by Kogge in [51], with the extension that the execution phase of instructions allows multiple register accesses and ALU operations. These operations may span multiple pipeline stages. For simplicity, we assume a pipelined machine with one-cycle stage latency throughout this section. With this assumption, a micro-operation of an instruction executed at its *C*'th cycle belongs to the *C*'th stage of the pipeline. The generalized case being stage latency of multiple cycles will be discussed in Section 6.3.

6.1.1 Definitions

To make the following discussion valid, we assume that a pair of dependent instructions travelling along the pipeline closely enough such that the pipeline structure will interfere with the dependencies of the pair.

An inter-instruction dependency in a pipeline structure can be described in terms of the triple (P, R_p, R_s). R_p and R_s are the register access patterns (read/write) of the preceding and succeeding instruction, respectively. The conventional classification of dependencies is encapsulated by this pair of parameters: data (R_p =write, R_s =read; or, read after write), anti (R_p =read, R_s =write; or, write after read), and output (R_p =write, R_s =write; or, write after write) dependencies. On the other hand, P describes the relative position of register accesses by a dependent instruction pair in a pipeline structure. The possible values of P are: forward, backward, and stationary. Thus P provides a classification of dependencies with a pipeline structure's point of view Figure 6.1 (a) shows the relative positions for the register accesses of preceding and succeeding instructions.

Suppose the instruction `instA` accesses register X at C_a'th cycle (at C_a'th stage), and the instruction `instB` accesses register X at C_b'th cycle (at C_b'th stage), with $C_a < C_b$. Now we define the first two classes of dependencies with respect to the pair (C_a, C_b) and the precedence relationship of instruction pairs: forward and backward

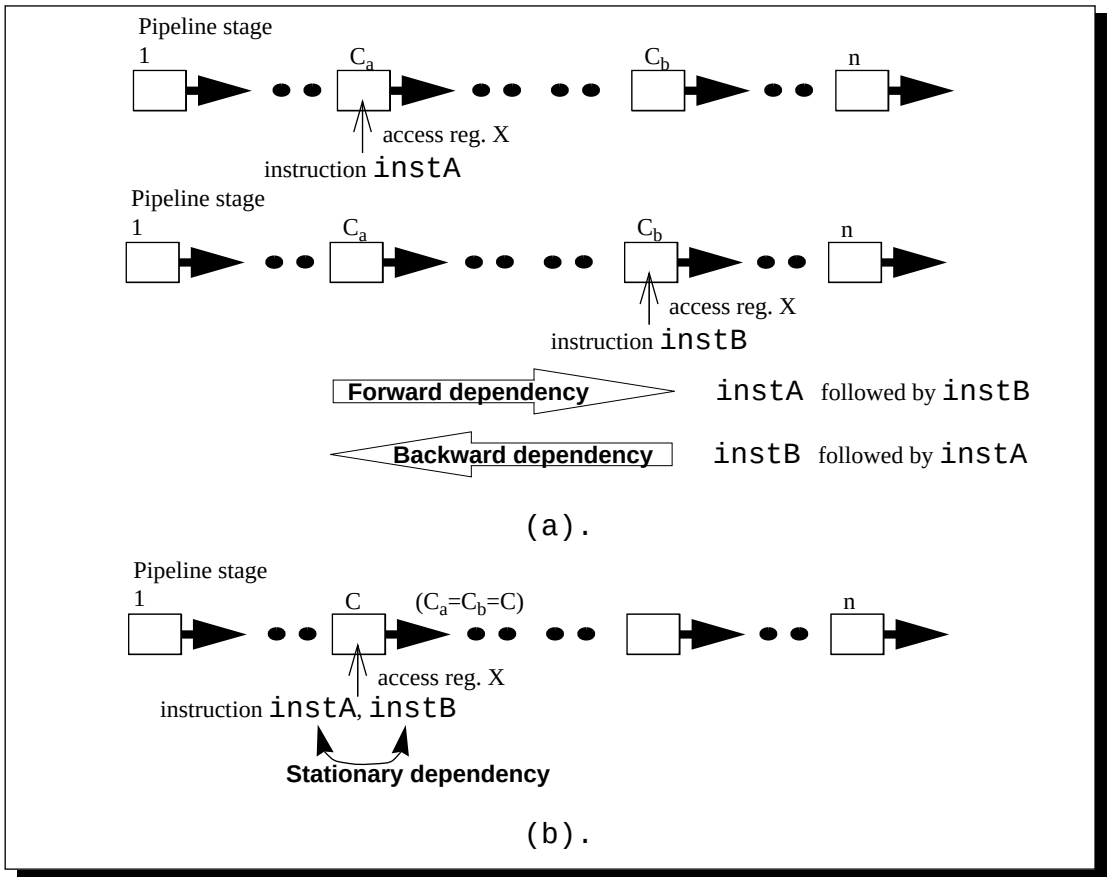


Figure 6.1 Forward, backward and stationary dependencies and pipeline stages

dependency. A *forward dependency* happens when a preceding instruction accesses a register at an earlier stage and a succeeding instruction accesses the same register at a latter stage such as the `instA-instB` pair (`instA` followed by `instB`); a *backward dependency* happens when a preceding instruction accesses a register at a latter stage and a succeeding instruction accesses the same register at an earlier stage such as the `instB-instA` pair (`instB` followed by `instA`). Note that both forward and backward dependencies potentially co-exist in the hardware for any pair of instructions that access the same register at different stages (except the read-read case). The actual direction of the dependency (forward or backward) in an application program is determined by the relative precedence relationship of the instruction pair; for example, the `instA-instB` pair has a forward dependency and the `instB-instA` pair has a backward dependency.

The third class of dependency is *stationary dependency* which happens when the preceding and succeeding instructions access the same register at the same stage ($C_a = C_b$) as shown in Figure 6.1 (b).

Please note that our definition of inter-instruction dependency is different from the definition in [51] in that our dependency is defined based on the access of a single register, as opposed to a set of registers. In [51], for every instruction, a *domain* and a *range* are defined as the set of registers that the instruction reads and writes, respectively. A pair of instructions exhibits an inter-instruction dependency as long as their domains or ranges have some overlap. This definition is insufficient since a single instruction may access more than one register, resulting in different types of pipeline hazards, each of which requires a different resolution. Therefore, a pair of dependent instructions may be involved in more than one class of dependency. With our classification, these hazards can be differentiated such that each dependency can be separately resolved.

The forward, backward and stationary dependencies can be further refined into nine types. By applying the conventional classification to each class of dependency, we have forward data, forward anti, forward output, backward data, backward anti, backward output, stationary data, stationary anti, and stationary output dependency. Table 6.1 summarizes these nine types of dependencies.

The preceding instruction accesses register X at cycle C_p The succeeding instruction accesses register X at cycle C_s	Types of register accesses		
	Read after Write (R_p=write, R_s=read)	Write after Read (R_p=read, R_s=write)	Write after Write (R_p=write, R_s=write)
$C_p < C_s$ (P = forward)	forward data dependency	forward anti dependency	forward output dependency
$C_p > C_s$ (P = backward)	backward data dependency	backward anti dependency	backward output dependency
$C_p = C_s$ (P = stationary)	stationary data dependency	stationary anti dependency	stationary output dependency

Table 6.1 Inter-instruction dependencies for pipelined instruction set processors

6.1.2 Pipeline hazards and inter-instruction dependencies

A pair of instructions exhibiting a backward dependency, when travelling closely in a pipeline, will cause a pipeline hazard because when the succeeding instruction reaches the stage where it accesses a register, the preceding instruction hasn't arrived at the stage (a latter stage) where it accesses the same register (as the `instB-instA` pair in Figure 6.1). For example, a 'delayed load' instruction immediately followed by another instruction that uses the loaded data in an earlier stage results in a pipeline hazard involving a backward dependency.

A forward dependency does not cause any pipeline hazard. The `instA-instB` pair in Figure 6.1 is an example of a forward dependency. Instruction `instA` accesses register X at stage C_a, leaving enough time (C_b - C_a +1 cycles) for the succeeding instruction `instB` to reach stage C_b to access X. However, even though a forward dependency does not cause any pipeline hazard, properly handling it may eliminate the associated backward dependency (for example, the `instB-instA` pair in Figure 6.1). We will further explain this issue in Section 6.2.

The stationary dependency does not cause any pipeline hazard, nor does it interfere with other classes of dependencies. The succeeding instruction can access the same register in the next cycle right after the preceding instruction's access. Instructions exhibiting stationary dependencies never access the same register simultaneously. There is no delay slot required for instruction pairs with stationary dependencies. Therefore, the stationary dependency is the most desired way of handling inter-instruction dependency in the pipeline design. This translates to a design goal that requires accesses to the same register in different instructions be scheduled to the same pipeline stage. However, this goal might not be achievable due to other design constraints. For example, aligning register accesses to the same pipeline stage may effectively lengthen the critical path.

When a stationary dependency can not be preserved, forward and backward dependencies occur, which necessitate some forms of resolution to ensure the proper behavior.

In the following section, we will present hardware and software resolutions for pipeline hazards caused by forward and backward dependencies.

6.2 Hardware/software Resolutions for Pipeline Hazards

The most powerful and practical way of resolving data dependent pipeline hazards is a data-flow like approach, such as Tomasulo's approach [76]. Data are assigned tags, and a data flow engine, such as the reservation station and its supporting circuitry in Tomasulo's approach, is used to track the relationships among data. In this paper, we consider an alternative approach employing relatively simple hardware and software techniques. An advantage of this approach is that it takes advantages of the classification of dependencies we presented in the previous section. We first summarize these resolution techniques in Section 6.2.1 and relate them to pipeline hazards in Section 6.2.2.

6.2.1 General hardware and software resolution strategies

Hardware resolution

The most straightforward way to resolve hazards in hardware is to use additional registers. Two types of additional registers can be employed: forwarding and duplicate registers.

Forwarding registers carry the data along with the instruction stream in the pipeline. For example, in Figure 6.2 a datum D is written to register X by instruction `instA` in stage C_2 at time `t`=1, and is forwarded along the pipeline via forwarding registers (X_{f1}, X_{f2}, X_{f3}) until D arrives at stage C_5 at time `t`=4. The datum moves along the pipeline synchronously with `instA`. The advantage of forwarding is that as soon as the current datum of register X is forwarded to next stage, X is free for more data. This is

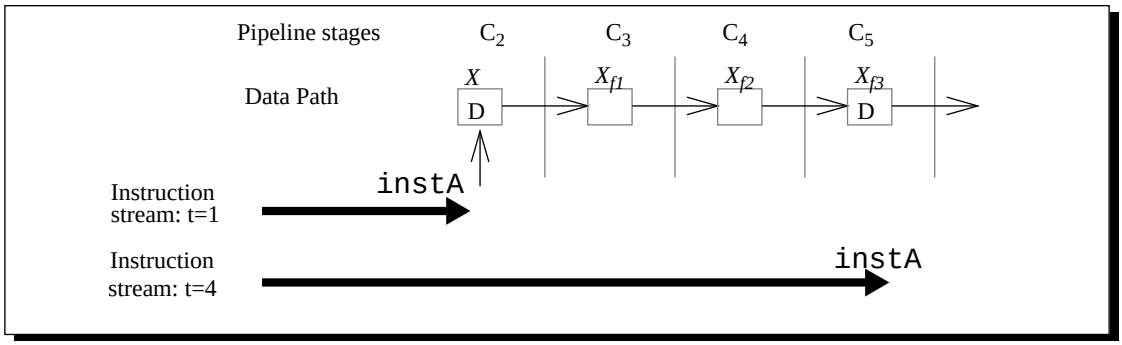


Figure 6.2 Hardware resolution: forwarding registers (X_{f1}, X_{f2}, X_{f3})

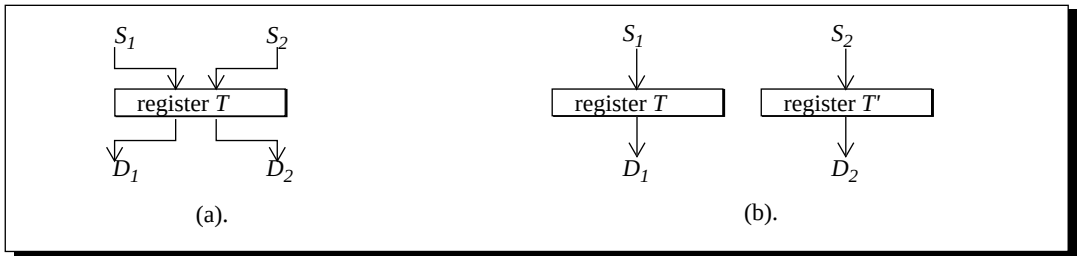


Figure 6.3 Hardware resolution: duplicate registers

analogous to adding extra latches (delays) in a pipeline in order to improve the system throughput [65].

Duplicate registers release the burden of temporary registers. In Figure 6.3 (a), a temporary register T connects two sources S_1 and S_2 and two destinations D_1 and D_2. There are four possible connection patterns. All connections are mutually exclusive, with T being the bottleneck of the data traffic. Suppose that the actual connections to be established by T are S_1->D_1 and S_2->D_2, and better performance will be achieved when these two connections can be made concurrently. Adding a duplicate register T', to create one additional data path, will ease the traffic and improve the performance as shown in Figure 6.3 (b).

Software resolution

Instruction reordering is the major technique used in compiler back-ends to resolve pipeline hazards. The desired behavior of an instruction stream may be distorted

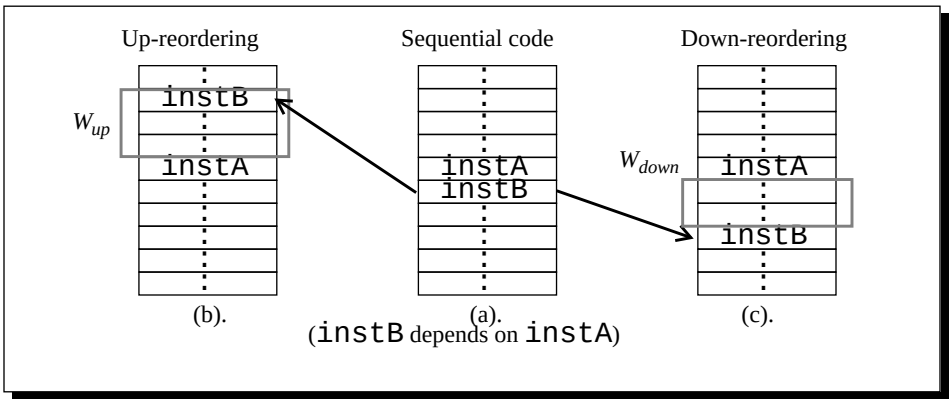


Figure 6.4 Software resolution: instruction reordering

by pipelining. Instruction reordering restores the desired sequential semantics of an instruction stream by reordering the sequence of instructions.

There are two directions in reordering: up and down reordering. In Figure 6.4 (a) is a sequential code where instruction `instB` follows and depends on `instA`. The cases (b) and (c) of Figure 6.4 are the reordered codes for two different pipeline structures. In case (b) the instruction `instB` is moved up and ahead of `instA`, whereas in case (c) `instB` is moved down and apart from `instA`. There are usually constraints (windows) to these movements: W_{up} being the maximal numbers of slots `instB` can be moved ahead of `instA` and W_{down} being the minimal number of slots `instB` has to be moved down from `instA`. For the example in Figure 6.4, W_{up} and W_{down} are three and two, respectively. We define W_{up} and W_{down} as the *reorder distance* for up and down reordering, respectively.

6.2.2 Inter-instruction dependencies and their applicable resolution

In this subsection we provide applicable hardware and software resolution strategies to pipeline hazards caused by forward and backward dependencies. As pointed out in Section 6.1.2, the stationary dependencies do not cause any hazard. Therefore, they do not require any resolution. For ease of discussion, we will use the same pipeline archi-

tecture and instruction pairs in Figure 6.1 as an example, assuming stage latency of one cycle. The generalized case where a pipeline stage takes multiple cycles will be provided in Section 6.3.

Forward dependencies

A forward dependency is the case of the `instA-instB` pair (`instA` followed by `instB`) (Figure 6.1). As mentioned in Section 6.1.2, a forward dependency does not cause pipeline hazards; however, some resolutions can be optionally applied to improve the system performance.

Forward data dependency

There are two ways to resolve this type of dependency: forwarding registers or up-reordering.

First, one forward register per stage can be allocated to stages C_{a+1} to C_{b-1} (totally $C_b - C_a - 1$ forward registers). A side effect of this approach is that the associated backward anti dependency (the `instB-instA` pair) is automatically resolved. This approach is preferable in the case where both `instA-instB` and `instB-instA` pairs happen very frequently in the application programs.

Secondly, instead of hardware resolution, one can choose to optionally move instruction `instB` ahead of `instA` (up-reordering) at most W_{up} ($W_{up}=C_b - C_a - 1$) slots (cycles). This has the advantage of hiding the possible delay slots associated with the instruction `instB` (for example, `instB` has a ‘delayed’ operation to other register *Y*), at the cost of longer compilation time in the compiler back-end.

Forward anti dependency

The applicable resolution is the up-reordering in the compiler back-end as described previously with $W_{up}=C_b - C_a$ (assuming the register is master-slaved).

Forward output dependency

There are two ways of resolving this type of dependency: duplicate registers and up-reordering.

Duplicate registers eliminate the forward output dependency such that `instA` and `instB` become independent instructions. The side effect is that the backward output dependency (the `instB-instA` pair) is eliminated as well. Forward output dependencies can also be resolved by the optional up-reordering in the compiler back-end as described previously with $W_{up}=C_b-C_a-1$.

Backward dependencies

A backward dependency is the case of the `instB-instA` pair (`instB` followed by `instA`). There is a software resolution, down-reordering, available for all backward dependencies. The dependent instruction `instA` has to be moved away, downward from its predecessor `instB` for at least C_b-C_a+1 slots ($W_{down}=C_b-C_a+1$) to ensure that `instA` access the target register after `instB`'s access. The compiler back-end can fill in these slots with `nop`s or reorder other independent instructions into these slots.

The backward output dependency can also be resolved in hardware with the same technique, duplicate registers, as in the case of forward output dependency. The side effect of the hardware resolution is that the associated forward output dependency is automatically resolved. Circular dependency check has to be performed before a duplicate register can be inserted. Unfortunately, there is no hardware resolution for backward data and anti dependencies.

6.3 Summary and Extension

Here we summarize the resolution strategies for inter-instruction dependencies in Table 6.2. In this table we extend the resolution strategies for pipeline machines with

multi-cycle stage latency: every instruction spends *S* cycles in a stage before it advances to next stage. Therefore, a micro-operation executed at the *C'th* cycle of its execution path will be executed at the `mod(`C/S`)+1`'th cycle of the $\lceil C/S \rceil$'th stage. In this table it is assumed that `instA` and `instB` access the same register at the C_a'th and C_b'th cycles of their execution paths, respectively. Please note that the constant *M* is used to adjust for the case of master-slave registers.

The first column contains the six classes of inter-instruction dependencies that require resolution. (Remember that the stationary dependencies do not cause pipeline hazards.) For each class of dependency, a dependent instruction pair taken from Figure 6.1 is listed as an example. Applicable hardware and software resolution strategies are listed in the second and third columns for each class of dependency. In the second column we also list the side effects of hardware resolutions. Note that forward data and forward/backward output dependencies have both hardware and software resolution candidates available, which allow tradeoff between hardware and software. The tradeoff can be based on the characteristics of the application domain (the frequency of the dependency in the applications) and the time/space complexities of the compiler backend. We will present a way to perform the tradeoff analysis in Section 7.1, and show examples in Section 8.4.

Inter-instruction dependency	Hardware resolution	Software resolution
Forward data (instA-instB)	• Forward registers total number of forward registers: $\lceil (C_b - C_a + 1 - M^*)/S \rceil - 1$ • Side effect of h/w resolution: backward anti dependency (instB-instA) is resolved	• Optional up-reordering: $W_{up} = \lceil (C_b - C_a + 1 - M^*)/S \rceil - 1$
Forward anti (instA-instB)	• n/a	• Optional up-reordering $W_{up} = \lceil (C_b - C_a + M^*)/S \rceil - 1$
Forward output (instA-instB)	• Duplicate register • Side effect of h/w resolution: backward output dependency (instB-instA) is resolved	• Optional up-reordering: $W_{up} = \lceil (C_b - C_a)/S \rceil - 1$
Backward data (instB-instA)	• n/a	• Down-reordering $W_{down} = \lfloor (C_b - C_a)/S \rfloor$
Backward anti (instB-instA)	• n/a	• Down-reordering $W_{down} = \lfloor (C_b - C_a - M^*)/S \rfloor$
Backward output (instB-instA)	• Duplicate register • Side effect of h/w resolution: forward output dependency (instA-instB) is resolved	• Down-reordering $W_{down} = \lfloor (C_b - C_a)/S \rfloor$

Table 6.2 Hardware/Software resolution strategies for inter-instruction dependencies that are not hazard-free

*. M= 1 for master-slave registers; M=0 otherwise.

Chapter 7 Pipeline Hazard Resolution

This chapter describes, in Section 7.1, the design procedure of the third phase (pipeline hazard resolution) of PIPER, which utilizes the extended taxonomy of inter-instruction dependencies and their hardware/software resolutions presented in Section 6. In Section 7.2, models of performance and cost are developed for the tradeoff of hardware and software.

7.1 Procedure for Pipeline Hazard Resolution

The pipeline synthesis process of PIPER consists of three phases: 1) pipeline assignment, 2) pipeline hazard resolution, and 3) resource allocation. The first phase, pipeline scheduling, assigns micro-operations into pipeline stages. Pipeline hazards may be introduced by the pipeline scheduler in highly pipelined cases. These hazards are resolved in the second phase. In this phase, application benchmarks are used to evaluate the design choices. At the last phase, the hardware resources for the data path are allocated, producing a pipelined RTL level design.

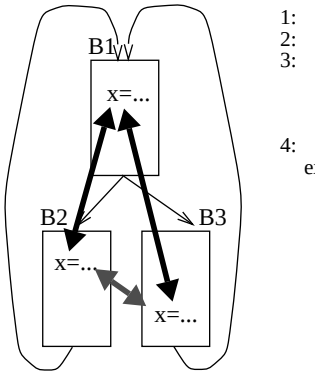
The pipeline hazard resolution phase takes as input a pipelined schedule, and outputs a set of hardware/software resolutions. This is accomplished in three steps. These steps are described in the following subsections.

7.1.1 Analysis of inter-instruction dependencies

In the first step, inter-instruction dependencies in the given pipelined schedule are identified. The inter-instruction dependencies appear as the *inter-iteration* dependencies in the pipelined schedule (pipelined loop body). There are two types of inter-iteration dependencies we are interested: *in-trace* and *cross-trace*. The in-trace dependencies are the dependencies that lie in the same execution trace within a single iteration, i.e.,

dependencies that can be detected without unrolling the loop. On the other hand, the cross-trace dependencies are those that lie across different execution traces within a single iteration, i.e., those that can be detected only when the loops are unrolled. For example, Figure 7.1 (a) shows a schedule consisting of three basic blocks (`B1`, `B2`, `B3`). The root block `B1` conditionally branches to block `B2` or `B3`. Blocks `B2` and `B3` are exclusive blocks and loop back to `B1`. There is a write to register `X` in every block. The two dark bi-directional arcs connecting register `X` accesses in block `B1` and `B2`, `B1` and `B3`, respectively, are in-trace dependencies, while the grey bi-directional arc connecting register `X` accesses in block `B2` and `B3` is a cross-trace dependency.

In Figure 7.1 (b) we present an algorithm to identify both in-trace and cross-trace dependencies without actually unrolling the loop. In the first step, global data flow analysis is performed on the loop body to identify the in-trace dependencies. While traversing through basic blocks, the analyzer records the earliest and latest read/write for each register, each exclusive block (such as `B2` and `B3` in Figure 7.1 (a)). In the second step, for each register and each set of exclusive blocks[1], cross-trace dependencies are gener-



(a).

1: GIVEN: a pipelined schedule
2: OUTPUT: in-trace and cross-trace dependencies
3: Perform a global data flow analysis for the loop body in the schedule:
(1). identify any in-trace dependency;
(2). for each basic block belonging to a branch (exclusive blocks):
record the earliest/latest read and write for each register
4: Perform pairing between earliest/latest register accesses in basic blocks for each set of exclusive blocks:
(1). *N*= number of exclusive blocks;
(2). label exclusive blocks: 1 to *N*;
(3). for *i*= *1* to *N* do
for each register in exclusive block *i*
pair the earliest read with earliest and latest write in blocks *i* to *N*
pair the earliest write with earliest and latest read/write in blocks *i* to *N*
pair the latest read with earliest and latest write in blocks *i* to *N*
pair the latest write with earliest and latest read/write in blocks *i* to *N*
report these pairs as cross-trace dependencies

(b).

Figure 7.1 (a). in-trace and cross-trace dependencies;
(b). algorithm for in/cross-trace dependency analysis

ated. They are formed by pairing the earliest/latest register read/write of one block with the earliest/latest register read/write of the other block, for every pair of exclusive blocks within the set.

7.1.2 Generation and weight assignment of resolution candidates

For each inter-instruction dependency identified in the first step, all possible hardware and software resolutions are generated in the second step, according to Table 6.2 on page 99. Some inter-instruction dependencies may have both hardware and software resolutions available. For such cases, we assign weights to hardware resolution candidates to help the designer to select the appropriate resolutions. In our current implementation, the weight is derived from the frequency in the application benchmarks and the reorder distance of the dependency that the hardware is to resolve. The following equation defines the weights. W_i is the weight assigned to the hardware resolution i;

$$W_i = \sum_{InstPair_i} Freq(InstPair_i) \times Dist(InstPair_i)$$

$InstPair_i$ is the instruction pair that contains the dependency the hardware is to resolve. $Freq(InstPair_i)$ is the frequency of the instruction pair $InstPair_i$ in the benchmark. $Dist(InstPair_i)$ is the reorder distance of the instruction pair $InstPair_i$ due to the dependency if it were not resolved by the hardware. Since a hardware resolution may resolve multiple dependencies (instruction pairs), the weight is calculated as a summation of the product $Freq(InstPair_i)Dist(InstPair_i)$ over all related instruction pairs. If application benchmarks are not available, an equal frequency is assumed. This equation intends to measure the hardware utilization and effectiveness of eliminating instruction reordering. However, the limitation of this simple model is that it does not consider the interaction

1. There may be more than one set of exclusive blocks. A set of exclusive blocks consists of blocks that are exclusive to each other.

1: GIVEN: selected hardware resolutions, software resolution candidates (reordering constraints), and inter-instruction dependencies
2: OUTPUT: software resolutions

3: For each hardware resolution
(1). delete the dependencies that is related to the hardware resolution;
(2). delete the software resolutions which belong to these dependencies;
(3). if the hardware resolution has a side effect on the related dependency, delete that dependency and its software resolutions

4: For the remaining dependencies, perform a merge process on their software resolutions (reordering constraints). A reordering constraint is a record with the format:
`ro(instruction1, instruction2, displacement)`
repeat the following until no further change can be made:
(1). if `ro(i1,i2,d1)` and `ro(i1,i2,d2)` exist and `d1` > `d2`, delete `ro(i1,i2,d2)`;
(2). if `ro(allInsts,i2,d)` exists, delete all `ro(ix,i2,dx)` with `dx` < `d`
(3). if `ro(i1,allInsts,d)` exists, delete all `ro(i1,ix,dx)` with dx < d

Figure 7.2 Algorithm for the generation of software resolutions

between resolutions of different dependencies. We will examine this limitation in the example section.

7.1.3 Generation of final resolutions

After the designer has selected the desired hardware resolutions, the software resolutions can be obtained. Figure 7.2 presents the algorithm for the generation of software resolutions from the candidates. In the first step, the side effects of the selected hardware resolutions are examined. As described in Table 6.2 on page 99, in addition to resolving the forward (backward) dependency it involves, a hardware resolution has a side effect of automatically resolving the associated backward (forward) dependency as well. Therefore, reordering constraints that are introduced by these dependencies are deleted from the software resolutions. In the second step, a merge process is performed on the remaining software resolutions. The purpose of this merge process is to remove the software resolutions that can be covered by others. For example, for down-reordering, suppose both `ro(add,load,3)` (three delay slots between `add` followed by `load`) and `ro(add,load,4)` exist, then the former can be deleted from the resolutions since it can be covered by the later constraint. The second step in Figure 7.2 is shown for the

down-reordering case. Up-reordering constraints can be obtained by interchanging '>' and '<' operation in the second step.

7.2 Performance and Cost Modelling

We characterize the synthesized design by the performance and cost of the hardware and its supporting software (the reorderer). The important metrics are peak/run-time performance and cost of hardware, and time/space complexities of compilers. With these metrics we are able to provide the designers with a broader and finer design space which spans both the hardware and the software. This capability of modeling the software and hardware components of complete systems is essential in extending the application scope of the design automation systems into embedded systems design.

7.2.1 The performance estimation of hardware

Two performance metrics are used to evaluate designs: maximal and run-time performances. The maximal performance can be obtained when the instruction latency and the clock rate are known. It does not vary with the applications. However, the run-time performance depends on the characteristics of the applications. The instruction trace expansion due to inter-instruction dependencies and pipeline hazards degrades the run-time performance of a deeply pipelined processor from its maximal performance. Consider a fragment of instructions compiled for non-pipelined machine in case (a) of Figure 7.3. Suppose that there is a synthesized pipelined machine which requires two delay slots between branch `brn` and its successive instruction of any type. In case (b) two `nop`'s are inserted between `brn` and the successive `sub r2`. The instruction trace is expanded by two `nop` slots which degrade the pipeline performance by two pipeline cycles. In case (c) the size of the `nop` slots is compressed from two to one by reordering an independent instruction `add r1` into the `nop` slots. PIPER uses a set of benchmarks

(instruction set programs) to estimate the run-time performance. Instead of repeating the phase of compiling and simulating the benchmarks for each pipeline implementation, PIPER applies the characteristics of simulation traces of benchmarks compiled for non-pipelined processors, and the characteristics of the reorder table to predict the run-time performance. With this approach, the run-time performance of different synthesized pipelined processors can be quickly obtained.

The instruction set simulator of ADAS provides a set of instruction benchmark characteristics. One of the characteristics used by PIPER is the pattern frequency of instruction pairs which is the frequency of the consecutive instruction pairs in the instruction trace. Shown in Figure 7.4 is the pattern frequency of a small instruction benchmark for the instruction set in Figure 7.4. For example, the first entry represents

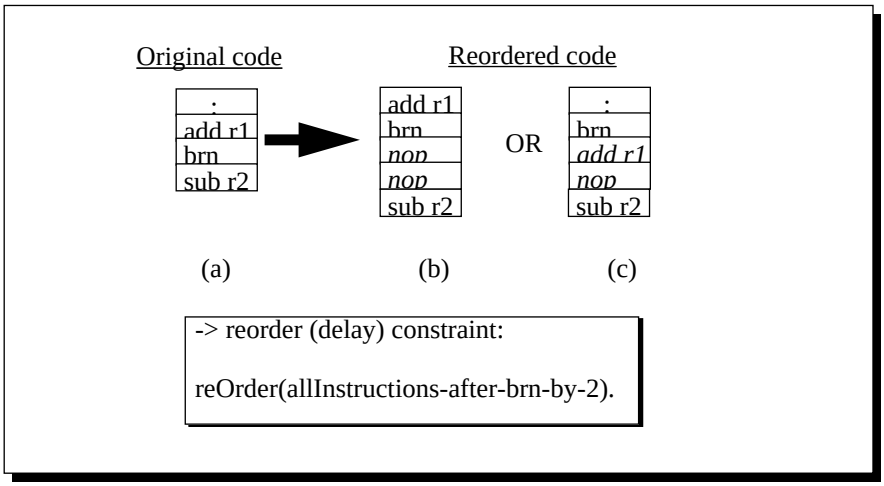


Figure 7.3 Instruction reordering

```
instruction_pattern_count([load,add],12,0.144578).
instruction_pattern_count([store,add],11,0.13253).
instruction_pattern_count([load,and],1,0.0120482).
instruction_pattern_count([add,brn],12,0.144578).
instruction_pattern_count([store,jump],11,0.13253).
instruction_pattern_count([brn,load],1,0.0120482).
instruction_pattern_count([jump,load],11,0.13253).
instruction_pattern_count([nop,load],1,0.0120482).
instruction_pattern_count([and,shr1],1,0.0120482).
instruction_pattern_count([add,store],11,0.13253).
instruction_pattern_count([brn,store],11,0.13253).
```

Figure 7.4 Pattern frequency of instruction pairs

that `load` immediately followed by `add` (the `load-add` pair) happens twelve times in the trace which has a frequency of 14.5%.

For each pipelined processor *p*, the run time performance metrics with respect to a benchmark can be approximated by the following equations:

$$Exp = \sum_j Dist_{p_j} \cdot (1 - C_j) \cdot R_j \cdot A + A$$

$$S_p = \frac{A \cdot M}{L_p \cdot Exp}$$

$$T_p = \frac{A}{L_p \cdot Exp}$$

where E_{xp} is the trace expansion due to the insertion of *nop's*, S_p is the speedup of pipelined processor *p* with respect to non-pipelined processor, and T_p is the throughput of pipelined processor *p*.

$Dist_{pj}$ is the number of *nop*'s (in the worst case) to be inserted between a consecutive instruction pattern *j* for pipelined processor *p*, R_j is frequency of the consecutive instruction pattern *j*, L_p is the instruction latency of pipelined processor *p*, *M* is the instruction cycle time of non-pipelined processor, *A* is the total number of instructions in the simulation trace, and C_j is the compression factor for consecutive instruction pattern *j* which means that, empirically, the percentage of the number of *nop* slots which can be filled by reordering instructions into the slots. C_j can be set to zero for the worst case analysis.

The product *AM* is the total number of cycles when the benchmark is executed on the non-pipelined processor while the product L_pE_{xp} is the total number of cycles executed on the pipelined processor *p*.

7.2.2 The cost estimation of hardware

We represent the cost of the hardware by the sizes of the data path and control path. The size of the data path is obtained by estimating the number of transistors in the data path. The wiring cost is not considered in our current version of PIPER. The control model adopted by PIPER is data stationary model. Each stage has its own controller. We implement these controllers as PLAs. Therefore, we derive the control path cost by summing up the estimated PLA size of each stage.

7.2.3 The time/space complexities of instruction set compilers

When comparing the time and space complexities of instruction set compilers for different pipeline implementations of the same instruction set architecture, we focus on the reorder phase of the compiler since the other phases of the compiler remain constant. We apply the characteristics of the reorder table to the equations of time and space complexities to derived the relative performance and cost of the reorderer, respectively. The possible characteristics of reorder table include the number of reorder entries (consecutive instruction patterns) E, the maximal safe instruction distance D_{max}, the minimal safe instruction distance D_{min}, and the average safe instruction distance D_{avg}.

The time/space complexities of the reorderer can be considered as the constraint imposed on the synthesis system by the software (the reorderer). For example, a reorderer with time complexity of E^3D_{max} implies that it is not feasible for a synthesis system to synthesize a design which produces a larger reorder table; however, a design with large safe instruction distance but a smaller reorder table may still be an acceptable design.

To demonstrate the idea, let's consider a simple reorderer which reads in the reorder table of size E, and then sweeps a window of size $2*D_{max}$ through the instructions to look for the target patterns, and then searches within the window for independent

instructions to be inserted into the *nop* slots. Assume that the table is kept in an array. Then for each iteration, the reorderer takes $O(E*D_{max}+D_{max}^2)$ time. At least, memory of size $O(E+D_{max})$ is necessary. Its time and space complexities may be expressed as:

$$\text{time complexity} = L \cdot D_{max} \cdot (E + D_{max})$$

$$\text{space complexity} = E + D_{max}$$

where L is the benchmark length. By substituting the variables for values, we can obtain the relative complexities. Similarly, we can derive the time/space complexities in terms of reorder table parameters for any reordering algorithms. This set of equations is kept in a system configuration file in PIPER and should be modified if the reordering technique used in the compiler is changed.

7.2.4 Exploration of the hardware/software tradeoff

Design decisions made by the pipeline scheduling and hazard resolution affect the performance and cost of hardware and the time/space complexities of the compiler back-end. Therefore, in addition to the design engine, there are a set of estimators and application benchmarks for the estimation of those effects.

As shown in Table 6.2, pipeline hazards caused by some inter-instruction dependencies may have both hardware and software resolution strategies available. Deciding which strategy to pick requires a tradeoff analysis. The hardware resolution has the advantage of resolving both forward and backward dependencies at the same time, at the cost of some additional registers and connections in the data path. On the other hand, the software resolution has the advantage of not complicating the hardware. However, it places a burden on the compiler back-end; the more reorder constraints, the longer the time to compile code.

The tradeoff analysis performed by PIPER is as follows. On the hardware side, the cost of hardware is represented by the estimated size of the processor including data path, forwarding and duplicate registers (if adopted), and control path. A set of application benchmarks is used to measure the performance of the hardware. The performance is estimated with the analytical approach described in Section 7.2.1.

On the software side, the time and space complexities of the compiler back-end are expressed as functions with parameters related to the characteristics of the reorder constraints such as the size of the reorder table, the maximal/minimal/average reorder distances in the table, etc. For example, a straightforward implementation of the reorderer (compiler back-end) described in [31] has time and space complexities proportional to the size of the reorder table (the number of reorder constraints). Therefore, when comparing the software cost of several possible designs, we evaluate, for each design, the time and space complexity functions by substituting the characteristics of its reorder constraints into the functions. In practice, the space complexity is of less concern since the space (static code size of the reorderer) varies just slightly for hardware designs with different sizes of reorder tables. Only the space taken up by the table varies, which is usually small when compared to the space of the algorithm.

The performance and cost of hardware and software are then taken together to evaluate the global "goodness" of the designs. The objective function used by PIPER for "goodness" is,

$$Goodness = P / C_{total} \qquad \text{EQ 7}$$

$$C_{total} = \alpha \cdot H + (1-\alpha) \cdot k \cdot S. \qquad \text{EQ 8}$$

H is the size of the hardware, S is the time complexity of the reorderer (representing the compilation difficulty), and k is a constant that adjusts the scales of H and S. The parameter α ($0 \le \alpha \le 1$) is used by the designer to assign the relative importance of hardware with respect to software (reorderer). C_{total} is a conceptual cost of the entire system

including hardware and software. *P* is the estimated performance in term of speedup with respect to the non-pipelined design. The *Goodness* is then defined as the "performance/cost" ratio of the entire system.

With the concurrent hardware/software approach and the tradeoff model, PIPER providers designers the capability of customizing designs for different application domains by selecting appropriate benchmarks and the weight constant α.

Chapter 8 Experiments

This chapter demonstrates the techniques developed in this dissertation. First, experiments of the instruction set design techniques in Section 5.1and Section 5.2 are given in Section 8.1. Second, experiments of simultaneous instruction set design and resource allocation are given in Section 8.2. Third, experiments of instruction set design and resource allocation for Prolog are given in Section 8.3. Finally, experiments of synthesizing microarchitectures and compiler backend interfaces are given in Section 8.4. The experiments were performed to show both the strength and limitation of our techniques.

The features of the conducted experiments include the following. (1) Both illustrative and practical examples are used for the experiments. (2) Throughout the experiments it is assumed that different microarchitectures have the same clock cycle time. Although more accurate estimation of the clock cycle time can be used, the simple assumption is adopted in the experiment to reduce the programming effort. (3) Objective and estimating functions are used to guide the design process and compare different designs. The goal of the experiments is to demonstrate how the techniques synthesize designs and explore design space, assuming the given objective and estimating functions are correct. Whether the given objective functions capture faithfully the designer's intention and whether estimating functions are accurate are not the concerns of the experiments. In the feature, the feedback mechanism described in Section 3.1 on page 32 will be implemented and used to improve the objective and estimating functions and design heuristics.

8.1 Instruction Set Design with Resources Allocated

This section demonstrates synthesis of application-specific instruction sets for architecture templates of which resources are given by the designer. The instruction sets are synthesized under the assumption that the given application benchmarks are the only software to be executed on the target processors. Therefore, only the instructions that support the required functionality of the given benchmarks are synthesized. This assumption is made to reflect the design requirement for embedded systems where one or a limited number of application benchmarks are run repeatedly. In the experiments, application-specific instruction sets are generated for each individual application. The purpose is to explore the variation of their architectural properties of several different application benchmarks by comparing performance and cost of their corresponding instruction sets.

Source-level (Prolog) application benchmarks are used for the experiments. They are compiled to assembly code by the Aquarius Prolog compiler [70]. A preprocessor maps the '`.r`' files of the assembly code to dependency graphs of micro-operations, which are the input to the synthesis algorithm described in Section 5.2 on page 73. The '`.r`' files are the machine level code assuming sequential, i.e., nonpipelined, execution[1].

As described in Chapter 5, the synthesis algorithm produces both the synthesized instruction set and assembly code for the given application benchmark. However, due to the lack of simulators for the synthesized instruction sets, it is difficult to obtain run time performance of the assembly code. At the moment, run time performance is approxi-

1. The last phase of the backend of the Aquarius Prolog compiler transforms the '`.r`' files into '`.ro`' files, which is the machine code ready to be executed on the 5-pipeline-stage VLSI-BAM microprocessor. Operations performed during the last phase includes insertion of `nop`'s, instruction reordering and some peephole optimization.

mated by assuming that each instruction in the assembly code get executed once. In the future, simulators will be automatically generated for the synthesized instruction sets.

8.1.1 A small example

In this example, we assumed the target architecture in Table 4.2 on page 59, the instruction field specification in Table 8.1, and the resource and delay specification in Table 4.3 on page 60 and Table 4.4 on page 60, respectively. The example used in this subsection is a small application which sets up a list of two elements in Prolog. It consists of 18 MOPs. Table 8.2 lists the MOPs and their dependencies. The `bf` clauses in the last row specify the *before* dependencies between MOPs. For example, `bf(1,4)` constrains that MOP 1 has to be scheduled in a time step earlier than MOP 4's. The `ctl(18)` clause specifies that the MOP 18 changes the control flow. Note that the control flow change has one cycle delay. We synthesized the 32-bit and 64-bit instruction sets, with the resource constraints <3R, 1W, 2M, 1F> and <6R, 4W, 4M, 4F>, respectively. The objective function used is EQ 9 with P=1.

$$Objective = (100/P)ln(C) + I \qquad EQ\ 9$$

Instruction Field Type	Number of bits
instruction word	32
opcode	6
register (R)	5
tag (T)	2
displacement (D)	16
immediate (I)	14
relation (<,=,>,≠) operator (OP)	2

Table 8.1 Bit width specification for some instruction field types

MOP ID	Type ID	RTLs*	MOP ID	Type ID	RTLs
1	rrait	r0 <- lst^(r1 + 0)	10	rrai	r1 <- r1 + 1
2	rit	r2 <- atm^36	11	rit	r2 <- atm^(-1)
3	mr	m(r1) <- r2	12	mr	m(r1) <- r2
4	rrai	r1 <- r1 + 1	13	rrai	r1 <- r1 + 1
5	rrait	r2 <- lst^(r1 + 1)	14	rrait	r3 <- var^(r1 + 0)
6	mr	m(r1) <- r2	15	mr	m(r1) <- r3
7	rrai	r1 <- r1 + 1	16	rrai	r1 <- r1 + 1
8	rit	r2 <- atm^37	17	rrai	r1 <- r1 + 1
9	mr	m(r1) <- r2	18	jd	pc <- pc +1024
Dependency bf: before ctl: control	bf(1,4). bf(2,3). bf(2,5). bf(3,5). bf(4,5). bf(4,6). bf(4,7).	bf(5,6). bf(5,7). bf(5,8). bf(6,8). bf(7,10). bf(7,9). bf(8,11).	bf(8,9). bf(9,11). bf(10,12). bf(10,13). bf(11,12). bf(13,14). bf(13,15).	bf(13,16). bf(14,15). bf(14,16). bf(16,17). ctl(18).	

Table 8.2 The MOPs and their dependencies of a list-creating application

*. bit width: tag=2, Immed=14

The synthesized 32-bit instruction set is listed in Table 8.3, consisting of four instructions. Note that two instructions `inst11` and `inst12` contain encoded fields, in order to satisfy the required 32-bit word constraint. This instruction set compiles the application into 12 cycles, as shown in Table 8.4. Note that time step 12 is the delay slot of `inst11` which changes the control flow. An independent instruction `inst12` is scheduled into time step 12 to make use of the delay slot.

Time Step	Compiled Code	Time Step	Compiled Code	Time Step	Compiled Code
1	inst13(r2, atm, 36)	5	inst12(r1, r2, 1)	9	inst12(r1, r2, 1)
2	inst14(r0, r1, lst, 0)	6	inst13(r2, atm, 37)	10	inst14(r3, r2, var, 0)
3	inst12(r1, r2, 1)	7	inst12(r1, r2, 1)	11	**inst11(r1, 1024)**
4	inst14(r2, r1, lst, 1)	8	inst13(r2, atm, -1)	12 (delay slot)	inst12(r1, r3, 1)

Table 8.4 Compiled code with the 32-bit instruction set

Instruction name	Instruction fields	RTLs	MOP type ID*	Encoded fields*
inst11	R_1, D	pc <- pc + D; R_1 <- R_2 + I	jd, rrai	I=1, R_1=R_2
inst12	R_1, R_2, I	m(R_1) <- R_2; R_3 <- R_4 + I	mr, rrai	R_1=R_3=R_4
inst13	R, T, I	R <- T ^ I	rit	
inst14	R_1, R_2, T, I	R_1 <- T ^ (R_2 + I)	rrait	

Table 8.3 32-bit instruction set

*. The right two columns specify the binary tuples for the corresponding instructions.

Table 8.5 lists the 64-bit instruction sets, consisting of five instructions. Most of the instructions have concurrent MOPs. Since 64 bits are wide enough to accommodate all instruction fields, there is no encoded field required in this instruction set. The compiled code (Table 8.6) consists of 9 cycles, which is 3 cycles less than the 32-bit one. Also note that the instruction `inst16` is scheduled to the delay slot of instruction `inst15` which changes the control flow.

Instruction name	Instruction fields	RTLs	MOP type ID	Encoded fields
inst15	R_1, R_2, R_3, R_4, D, I	pc <- pc + D; m(R_1) <- R_2; R_3 <- R_4 + I	jd, mr, rrai	
inst16	R_1, R_2, R_3, R_4, I	m(R_1) <- R_2; R_3 <- R_4 + I	mr, rrai	
inst17	R_1, R_2, R_3, T, I_1, I_2	R_1 <- T ^ I_1; R_2 <- R_3 + I_2	rit, rrai	
inst18	R_1, R_2, R_3, T_1, T_2, I_1, I_2	R_1 <- T_1 ^ I_1; R_2 <- T_2 ^ (R_3 + I_2)	rit, rrait	
inst14	R_1, R_2, T, I	R_1 <- T ^ (R_2 + I)	rrait	

Table 8.5 64-bit instruction set

Time Step	Compiled Code	Time Step	Compiled Code
1	inst18(r2, r0, r1, atm,lst, 36,0)	6	inst16(r1, r2, r1, r1, 1)
2	inst16(r1, r2, r1, r1, 1)	7	inst18(r2, r3, r1, atm, var, -1,0)
3	inst14(r2, r1, lst, 1)	8	**inst15**(r1, r2, r1, r1, 1024, 1)
4	inst16(r1, r2, r1, r1, 1)	9 (delay slot)	inst16(r1, r3, r1, r1, 1)
5	inst17(r2, r1, r1, atm, 37, 1)		

Table 8.6 Compiled code with the 64-bit instruction set

8.1.2 Prolog application benchmarks

In this subsection, experiments are presented to show the versatility and practicality of our tools by synthesizing instruction sets for some application benchmarks, with various design constraints and objective functions. Four benchmarks were selected from the Prolog Benchmark suite [30]. The benchmarks `con3` and `nreverse` are programs for list manipulation. The benchmark `query` is a program for database query. The benchmark `circuit` maps boolean equations into logic gates. The second column in Table 8.7 lists the characteristics of the benchmarks, including the numbers of MOPs, data-related dependencies, and control dependencies in the benchmarks. The number of

MOPs represents the size of the benchmark; the number of data-related dependencies is related to the degree of parallelism available within the benchmark; the number of control dependencies indicates the degree of the impact of the branch/jump delays on the benchmark.

We assumed that every basic block executes once. we assumed the target architecture in Table 4.2 on page 59 and the instruction field specification in Table 8.1 on page 113. The delay constraints for control and memory operations are one and zero, respectively. The experiment was conducted on a HP750 workstation with 256M memory.

For each benchmark, we synthesized its 32-bit, 48-bit, and 64-bit instruction sets, respectively. We were interested in how the instruction sets vary with bit widths. Table 8.7 lists the results, synthesized under the objective function with P=1 in EQ 9 on page 113. For all three benchmarks, as we had expected, the cycle decreases when the instruction word width increases. However, we observed a smaller gain in `nreverse` and `circuit`. This can be explained by their larger ratios of the number of data dependencies to the number of MOPs. Most of the MOPs depend on each other such that there is less parallelism available when packing MOPs into instructions.

In general, the size of the instruction set also increases when the instruction word width increases. This is due to the fact that wider words can accommodate more MOPs, resulting in richer and more powerful instructions. However, the 48-bit instruction sets are ‘embarrassing’ designs for `con3` and `nreverse`. Their instruction set sizes are larger, and their performance is worse than their 64-bit alternatives in compiling the benchmarks. The 48 bits are not wide enough for these benchmarks to accommodate the most frequent MOP patterns, for which 64 bits are sufficient. Therefore, the design process has to specialize the general forms of some powerful instructions into several dis-

tinct instructions by making fields implicit or unifying register ports, in order to satisfy the bit width constraint.

In the ‘Instruction set space’ column we examined the number of instruction candidates explored by the design process. The numbers, much larger than the final instruction sets, show that the design process was able to explore a rich design space for the best candidates while keeping the size of the design space manageable.

In the two right most columns we also list the run time and memory usage of our algorithm, which show that our tools were able to synthesize instructions for application benchmarks within reasonable time and consume a modest amount of memory.

Benchmark	# of MOPs, data dep., control dep.*	Instruction word width†	Design results			Performance of the algorithm	
			Cycle (C)	Instruction set size (I)	Instruction set space	Time (minutes)	Memory (MB)
con3	183, 136, 24	32	135	29	1275	56	2.1
		48	93	38	3733	59	2.7
		64	89	35	3277	48	2.7
nreverse	245, 395, 11	32	169	17	540	69	2.1
		48	157	23	772	57	2.0
		64	154	22	688	48	2.0
query	391, 185, 68	32	305	24	478	95	2.0
		48	215	32	1742	103	2.3
		64	204	39	1445	89	2.3
circuit	1725, 1077, 274	32	1406	40	1710	1358	3.4
		48	1361	25	1389	1722	4.6
		64	1360	24	1362	4726	5.9

Table 8.7 Results (Objective function = 100ln(C)+S)

*. The number of control dependencies is counted as the total number of branch/jump MOPs.
†. The hardware constraints are 3R, 1W, 2M, 1F for 32-bit instructions; 6R, 3W, 2M, 3F for 48-bit instructions; 8R, 4W, 32M, 4F for 64-bit instructions

In Table 8.8 we compared the synthesized 32-bit instruction sets for these benchmarks with the BAM instruction set, which was designed for the VLSI-BAM micro-pro-

cessor by the Aquarius Project at the University of California, Berkeley. The VLSI-BAM micro-processor has RISC-style instructions plus some powerful instructions to support efficient logic computation such as Prolog. The benchmarks were compiled with the BAM instruction set, and we measured the number of distinct instructions used (in the 'Instruction set size' column), and the number of cycles to execute the compiled code (in the 'Cycle' column). The programs were compiled by the Aquarius Prolog Compiler, with the post-phase optimization phase turned off[2]. The experiments show that the synthesized instruction sets produced more compact codes for all four benchmarks, with 10%, 5%, 17%, and 3% reduction in the code size, respectively. This was achieved at the cost of a small number of additional instructions (7, 1, and 2 for `con3`, `nreverse`, and `query`, respectively), except in `circuit` where 16 additional instructions are required. We then used the objective function to evaluate the global performance/cost tradeoffs for both instruction sets and found that in most cases (`con3`, `nreverse`, and `query`) the synthesized ones yield better results, as indicated in the 'Objective value' column (smaller values are better). It is possible to improve the result of `circuit` by adjusting the initial temperature and the cooling schedule in our future experiment. We also compared the hardware resources used by both instruction sets. They both use the same amount of resources, except in the `nreverse` case our synthesized instruction set uses one less register read port and one less memory port than BAM does. This experiment shows that ASIA is capable of competing with manually designed instruction sets within our collection of benchmarks. Further studies will be needed to investigate its competence in more general cases.

2. The post-phase optimization of the Aquarius Prolog Compiler alters the classic definition of the basic block. Due to the time limit, we were not able to modify our tools to accommodate such change.

Benchmark	Instruction set	Hardware resources*	Cycle (C)	Instruction set size (I)	Objective value (smaller is better)
con3	BAM†	3R, 1W, 2M, 1F	150	22	523
	ASIA‡	3R, 1W, 2M, 1F	135	29	520
nreverse	BAM	3R, 1W, 2M, 1F	178	16	534
	ASIA	**2R**, 1W, **1M**, 1F	169	17	530
query	BAM	3R, 1W, 2M, 1F	368	22	613
	ASIA	3R, 1W, 2M, 1F	305	24	596
circuit	BAM	3R, 1W, 2M, 1F	1453	24	752
	ASIA	3R, 1W, 2M, 1F	1406	40	764

Table 8.8 Performance comparison with a manually designed instruction set

*. The resource constraints given to ASIA is the same as in the VLSI-BAM processor: 3R, 1W, 2M, 1F.
†. BAM refers to the instruction set that was manually designed for the VLSI-BAM processor.
‡. ASIA refers to the instruction set synthesized by the tools (ASIA) reported in this paper.

Table 8.9 shows some interesting instructions synthesized for the benchmark `query`. They are selected from the 32-bit, 48-bit, and 64-bit instruction sets, respectively. For ease of illustration, we do not list the binary tuples for these instructions; instead, we describe the RTLs of these instructions directly. In the RTLs, the register sharing is indicated by using the same register index. Note that the 32-bit version of the instructions can be found in the BAM instruction set as well. This fact provides the BAM designers with more confidence about their instruction set, since some of the instructions that they considered ‘powerful’ retain their existence when the instruction set is designed by other independent designers (in this case, the ASIA design automation system). This observation suggests that ASIA, in addition to its original purpose (an automatic design tool), can be used as a verification tool for designers to verify their manually designed instruction sets as well.

Instruction word width	RTLs*	Meaning
32	$m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$	push†
	if (tf=1) { $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	conditional push†
	if (tag(R_1=T_1) { pc ← pc + D_1 }; if (tag(R_1=T_2) { pc ← pc + D_2 }	switch on tag†
48	tf ← R_1 OP. R_2; pc ← I	compute condition and jump
	$m(R_2) \leftarrow R_3$; if (tf=1) { $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	store and conditional push (with a shared register)
	$m(R_1) \leftarrow R_2$; $R_3 \leftarrow R_4+I$	store and add
64	$m(R_3) \leftarrow R_4$; if (tf=1) { $m(R_1) \leftarrow R_2$; $R_1 \leftarrow R_1+D$}	store and conditional push
	$m(R_1) \leftarrow R_2$; $R_3 \leftarrow R_4+ I$	store and add
	$R_1 \leftarrow T \wedge I$; $R_2 \leftarrow R_3 + I$	tag data and add

Table 8.9 Some synthesized instructions

*. Notations: 1). The RTLs in an instruction are executed simultaneously; 2). *tf*: a one bit latch which holds the truth value of a logic computation; 3). The operator '^' appends a tag to a value before the value is sent to a destination.
†. These three instructions can be found in the BAM instruction set.

Finally, Table 8.10 shows how the synthesized instruction sets vary with the objective functions. In this experiment we synthesized 32-bit instruction sets for the benchmark `query` with two versions of the objective function EQ 9 on page 113: one with P=1, another with P=5. The latter assigns less importance to the cycle count. Therefore, the tools focused on reducing the instruction set size, resulting in 7 instructions less, but 16 cycles more than the former case.

Objective function	Cycle (C)	Instruction set size (I
100•ln(C)+S.....P=1 in EQ 9	135	29
20•ln(C)+S.....P=5 in EQ 9	151	22

Table 8.10 Instruction variation for the benchmark `query` due to different objective functions

8.2 Instruction Set Design and Resource Allocation

The experiments conducted in this section are under the same setting in Section 8.1, except that the number of the hardware resources is not constrained and is left to the design system to allocate. In the experiments, application-specific instruction sets are synthesized, assembly code is generated and resources are allocated for each individual application. The purpose is to explore the variation of their architectural properties of several different application benchmarks by comparing the resulting instruction sets and resource usage.

We used the same MOP specification and timing parameters in Section 8.1 as the given architecture template. The bit width constraints for instruction fields is given in Table 8.1. The objective function can be an arbitrary complex function. To simplify the experiment, EQ 10 is used as the objective function for the experiment in this section. In the equation, C is the dynamic cycle count, I is the instruction set size, R is the number of register-file read ports, W is the number of register-file write ports, M is the number of memory ports, and A is the number of ALUs.

$$\text{Objective} = 100\ln(C) + I + 2R + 3W + 5M + 4A \qquad \text{EQ 10}$$

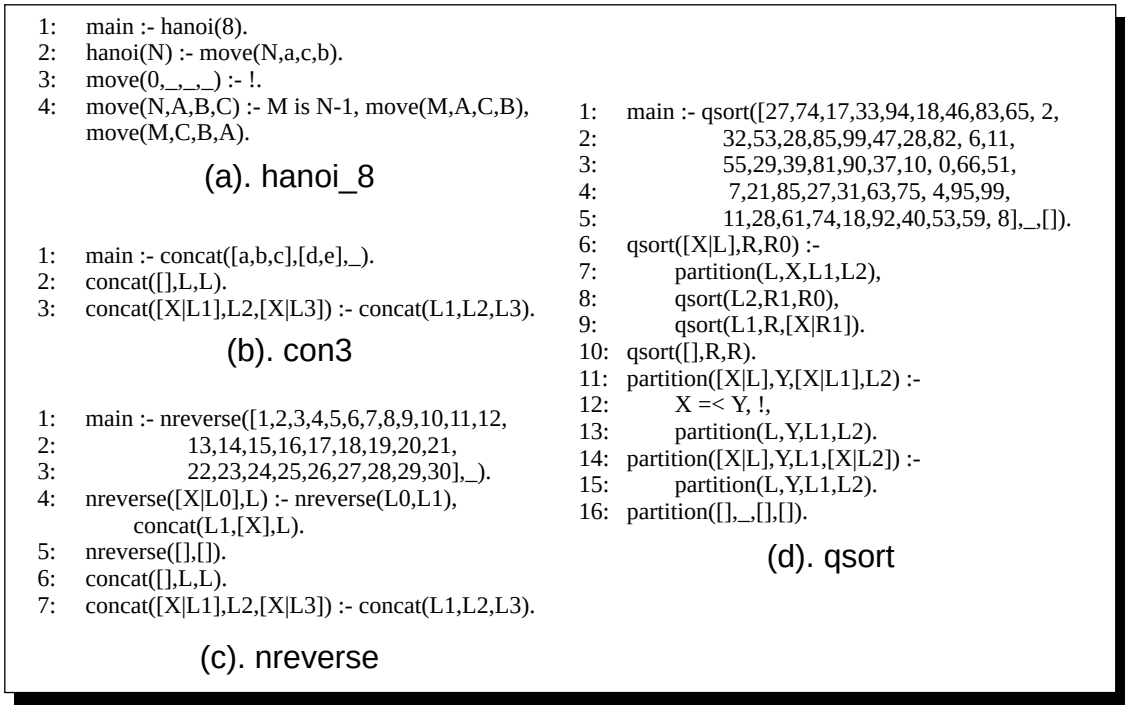
We assumed that every basic block executes once for every application. With this assumption, the *T* parameter in the objective function represents the static code size, instead of the execution time[3]. The number of movements tried at each temperature point is *5*(# of MOPs)*. The next temperature is 90% of the current temperature. The experiments were conducted on a HP 750 workstation with 256M memory.

Four symbolic applications were selected from the Prolog benchmark suite [30]. `hanoi_8` is the ‘hanoi’ problem solver. `con3` concatenates two strings into one string.

3. This assumption was due to the fact that our profile analyzer was not available at the moment such that we were not able to obtain the run time behavior, i.e., the execution counts of basic blocks.

nreverse reverses the order of the given string. Note that the predicate `concat/3` defined in `con3` is used as a subroutine in *nreverse* as well. `qsort` is the classic 'qsort' algorithm. Figure 8.1 lists the source code in Prolog. The `main` clauses are given by the user to represent the typical execution of the programs. Note that due to its built-in unification and backtracking support, Prolog gives more compact representation than C language does. The C representation is usually five to ten times larger than the Prolog representation. However, when compiled to intermediate representation, the hidden details in Prolog become explicit.

The results are given in Table 8.7. In the second column are some characteristics about the applications. The number of MOPs represents the size of the application. The numbers of data and control dependencies limit the available parallelism in the MOPs. The ratio #MOP/#Dep tries to measure the "average" parallelism. The columns under the header "Design results" are the outputs of the algorithm: the resource allocation,



```
1:  main :- hanoi(8).
2:  hanoi(N) :- move(N,a,c,b).
3:  move(0,_,_,_) :- !.
4:  move(N,A,B,C) :- M is N-1, move(M,A,C,B),
        move(M,C,B,A).
```

(a). hanoi_8

```
1:  main :- concat([a,b,c],[d,e],_).
2:  concat([],L,L).
3:  concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

(b). con3

```
1:  main :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,
2:                    13,14,15,16,17,18,19,20,21,
3:                    22,23,24,25,26,27,28,29,30],_).
4:  nreverse([X|L0],L) :- nreverse(L0,L1),
        concat(L1,[X],L).
5:  nreverse([],[]).
6:  concat([],L,L).
7:  concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

(c). nreverse

```
1:  main :- qsort([27,74,17,33,94,18,46,83,65, 2,
2:                 32,53,28,85,99,47,28,82, 6,11,
3:                 55,29,39,81,90,37,10, 0,66,51,
4:                  7,21,85,27,31,63,75, 4,95,99,
5:                 11,28,61,74,18,92,40,53,59, 8],_,[]).
6:  qsort([X|L],R,R0) :-
7:        partition(L,X,L1,L2),
8:        qsort(L2,R1,R0),
9:        qsort(L1,R,[X|R1]).
10: qsort([],R,R).
11: partition([X|L],Y,[X|L1],L2) :-
12:       X =< Y, !,
13:       partition(L,Y,L1,L2).
14: partition([X|L],Y,L1,[X|L2]) :-
15:       partition(L,Y,L1,L2).
16: partition([],_,[],[]).
```

(d). qsort

Figure 8.1 Application programs in Prolog

cycle counts of the application, the instruction set size. Under this header we also list the number of candidate instructions (in the “Inst. set space” column) and the number of microarchitecture configurations (in the “uArch space” column) explored by the algorithm. Finally, in the two right most columns we list the CPU run time and the memory usage of the algorithm.

Benchmark	# of MOPs, # of data dep., # of control dep.,* #MOP/#Dep.	Design results					Performance of the algorithm	
		Resource allocation†	Cycle (T)	Inst. set size (S)	Inst. set space	uArch space	Time (min.)	Memory (MB)
hanoi_8	51, 22, 9, 1.65	2R, 1W, 1M, 1A	37	18	126	6	9	1.7
con3	183, 136, 24, 1.14	3R, 1W, 2M, 2A	116	31	244	5	35	1.9
nreverse	430, 503, 36, 0.79	3R, 1W, 2M, 2A	306	24	275	6	181	2.4
qsort	744, 847, 65, 0.81	3R, 1W, 2M, 2A	523	27	114	7	492	2.8

Table 8.11 Synthesis results

*. The number of control dependencies is counted as the total number of branch/jump MOPs.
†. Notation: ‘R’=read port of register-file, ‘W’=write port of register-file, ‘M’=memory port, ‘F’=functional unit, and the value is the number of a particular hardware resource. For example, ‘2R’ means two read ports for register-file.

Several interesting architectural properties are revealed by the experiment. First, `hanoi_8` has the largest “average” parallelism among all applications as indicated by the #MOP/#Dep ratio, which implies that allocating more resources and instructions may make more sense to `hanoi_8` than to others. However, the results of the experiment contradict our expectation: `hanoi_8` gets least amount of resources and instructions. After examining the MOP graphs of applications in detail, we found that the seemingly abnormal design results are due the irregularity of the basic block structures. The variation of the basic block sizes in `hanoi_8` is smaller than the variations in others. In other words, the basic block structure of `hanoi_8` is more regular than others’. Therefore, the number of instructions (MOP patterns) required to map `hanoi_8` is less

than what are required for others. This explains the smaller size of `hanoi_8`'s instruction set. On the other hand, lots of the dependencies in the applications other than `hanoi_8` are located in some tiny basic blocks, leaving some large basic blocks with very few dependencies. Since in this experiment, our objective is to minimize the compiled code size, instead of the dynamic code size (run time), the advantage of large amount of parallelism existing in those large basic blocks was taken by the algorithm, and larger amount of resources were thus allocated to these applications. This explains why these applications get more resources than `hanoi_8` does. This observation suggests that, in addition to the simple measurement of "average" parallelism by the #MOP/#Dep ratio, some type of "locality" indicator, measuring the distribution of dependencies, is necessary to help the characterization of applications.

Second, as we have observed in the program listing, the major subroutine `concat` of the application *con3* is also in the application *nreverse*. Therefore, the former can be viewed as part of the latter. In other words, the latter is more "general" than the former. This suggests that there may exist some similarities and covering properties between the synthesized microarchitectures and instruction sets for these applications. The results show that both applications are allocated the same amount of resources. However, the instruction set size (31) of *con3* is larger than the one (24) of *nreverse*. The existence of many specialized, powerful instructions for *con3* can not be justified in *nreverse* since the frequencies of such special patterns decrease in a larger, more general environment. In Table 8.12 we compare some specialized, powerful instructions in *con3* with their corresponding ones in *nreverse*. Some instructions remain their existence, such as the ones in case 1, 2 and 3. Some disappear, such as the ones in case 4 and 5. Some lose their specialization, such as the one in case 6. On the other hand, most of the instructions in *nreverse* remain their existence in *con3*, in their original forms (20 out of 24) or specialized forms (3 out of 24), except the ones in case 6 and 7.

We found that the instruction in case 7 is used only once in *nreverse*. If we delete it from the instruction set, and map the two MOPs it contains with two simple instructions which already exist in the instruction set, then the objective value can be further reduced, resulting in a better solution. However, this was not done by our algorithm. The reason is that the chance of the MOPs being selected and displaced by the algorithm was very low since this pattern occurred only once in the application. To fix this problem, we can increase the number of movements tried at each temperature point at the cost of increased CPU time. On the other hand, we can introduce more powerful move operators such as “delete an instruction” or “delete an resource” to the algorithm at the cost of complicating the design heuristics and modification to the data structure, since the objects being moved is no longer just MOPs, the simple and local ones, but also instructions and resources, the complex and global ones.

Case	Instruction in the *con3* instruction set*	Corresponding instruction in the *nreverse* instruction set
1	m(**R_1**) ← **R_2**; m(**R_1**+1) ← **R_3**; **R_1** ← **R_1**+2	same
2	m(**R_1**) ← **R_2**; **R_1** ← **R_1**+**D**	same
3	if (tag(**R_1**=**T_1**) { pc ← pc + **D_1** }; if (tag(**R_1**=**T_2**) { pc ← pc + **D_2** }	same
4	tf ← **R_1** OP. **R_2**; m(**R_1**+0) ← **R_3**	not available
5	**R_1** ← pc; **R_2** ← **R_2**+ **I**	not available
6	if (tf=1) { m(**R_1**) ← **R_2**; **R_1** ← **R_1**+1}; m(**R_1**+0) ← **R_3**	if (tf=1) { m(**R_1**) ← **R_2**; **R_1** ← **R_1**+**D**}
7	not available	pc ← **I**; r31 ← pc; **R_1** ← **R_2**+**D**
8	not available	**R_1** ← **I**

Table 8.12 Comparison of instruction sets for `con3` and `nreverse`

*. Operands that have to be explicitly specified in the instruction fields are printed with bold-face.

Figure 8.2 illustrates how the objective value, resource allocation, instruction set size and cycle count varied during the annealing process for the application `hanoi_8`. The data points were sampled at the end of each temperature point: 30 temperature

points were computed in this application, and 255 movements were tried at each temperature point.

For comparison, we also synthesized the best microarchitecture and instruction set for `hanoi_8` using an iterative approach over a range of possible resource allocation.

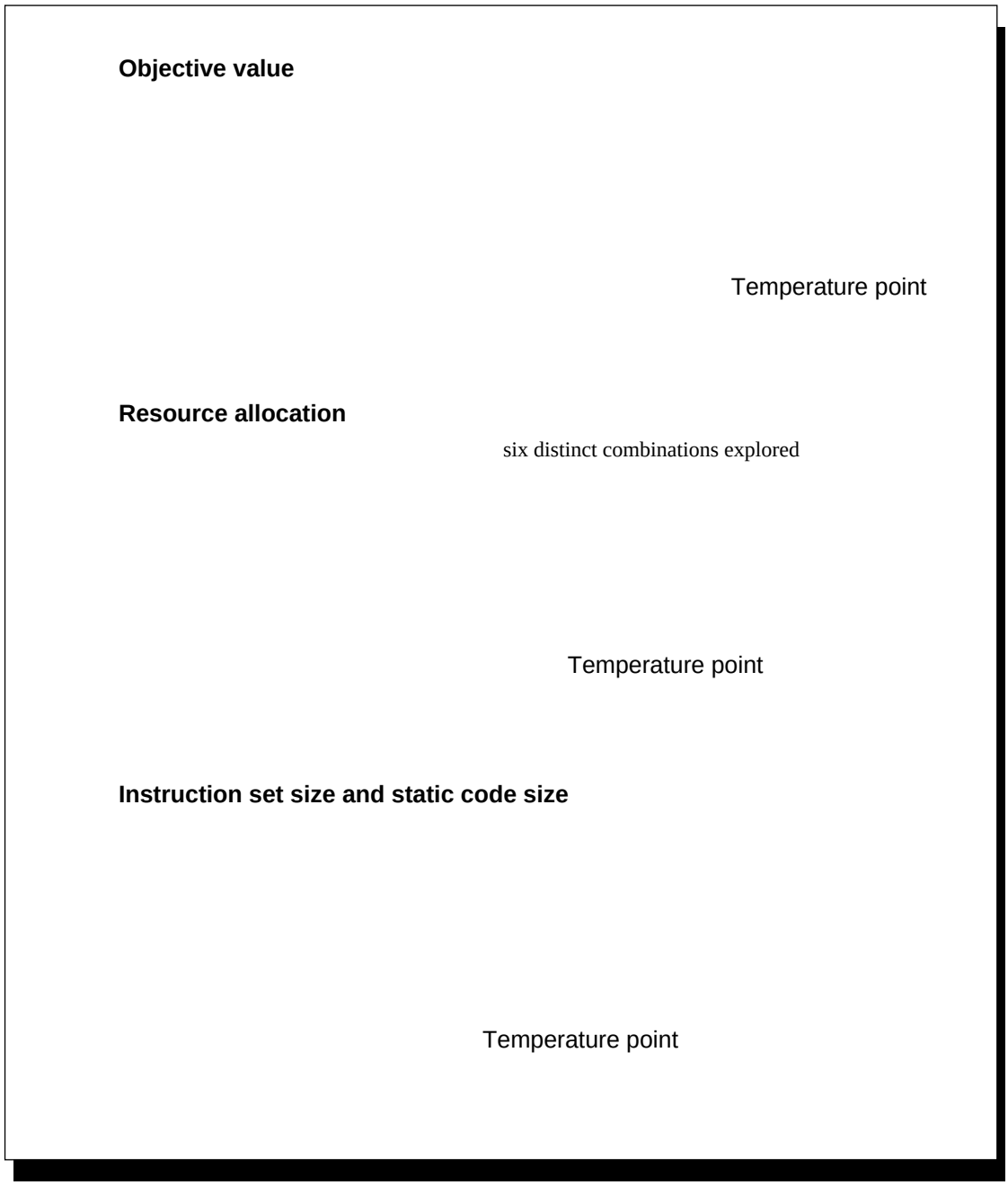


Figure 8.2 The simulated annealing process for the application `hanoi_8`

During each iteration, ASIA was configured to generate instruction sets and assembly code for the given resource allocation. Therefore, the most difficult task is to decide that how many combinations of resource allocation should be tried before we believe that sufficient design space has been explored. We used the six resource configurations explored in Figure 8.2 as the guide for the experiments. The results are shown in Table 8.13. The iterative approach obtained the same instruction set and hardware allocation as the integrated approach did in Table 8.7, but used almost 3 times longer CPU time. While the iterative approach has to conduct several complete runs on various resource configurations in order to find the best solution, the integrated approach finds the best solution much faster by dynamically switching between different resource configurations (as shown in Figure 8.2) such that infeasible design space can be pruned early in the search process. The results show that the integrated approach, in addition to the clarity in the problem formulation, is a significant performance improvement (about three times) over the iterative approach in solving the combined problem of instruction set design, microarchitecture design and code generation.

Given resource allocation	Design results				Performance of the algorithm	
	Cycle (C)	Inst. set size (S)	Inst. set space	Objective value	Time (min.)	Memory (MB)
3R, 2W, 2M, 2A	34	21	213	1105	6	1.7
3R, 1W, 2M, 2A	36	20	154	1099	7	1.7
3R, 1W, 1M, 1A	37	21	143	1062	5	1.7
2R, 2W, 1M, 1A	38	19	137	1069	6	1.7
2R, 1W, 1M, 2A	37	21	169	1073	5	1.7
2R, 1W, 1M, 1A	37	18	169	1047	5	1.7
TOTAL EXPERIMENT TIME					34	

Table 8.13 An iterative approach for instruction set design and resource allocation for the benchmark `hanoi_8`

8.3 Instruction Set Design and Resource Allocation for Prolog

The experiments in Section 8.1 and Section 8.2 were conducted for instruction set processors used in embedded systems. In an embedded environment, the size of the instruction set can be reduced by eliminating instructions whose functionality is not required in the applications, or can be substituted by other instructions.

On the other hand, the experiments conducted in this section are to synthesize a *complete* instruction set for Prolog that is customized toward a particular application. The application benchmark `con3` listed in Figure 8.1 on page 123 is used to customized the instruction set.

8.3.1 Complete segments for Prolog

To ensure completeness of the synthesized instruction set, special code segments, which were used by Holmer in Table 8.3 of his dissertation [36] for the same purpose, are included in the given application. The special code segments contain a subset of the VLSI-BAM instruction set which is known to be complete for Prolog execution. The complete segments are given in Figure 8.3, with each segment separated by the predicate `label`. The segments include basic operations required by Prolog: compare, branch, jump, load immediate, read the program counter, register-to-register move, register-to-register arithmetic, and other Prolog-specific operations (unification, deference, maximal, etc.).

The experiments are conducted with the same microarchitecture specification, and instruction bit width specification as in Section 8.2, and the objective function EQ 10 on page 122. We first synthesize an instruction set IS_{prolog} for the complete segments with ASIA. The result is as shown in Table 8.14. In the table, the first column are the identifiers of the synthesized instructions; the second column lists the binary tuples

for corresponding instructions; the third column shows the register transfers of the corresponding binary tuples; the fourth column shows the equivalent VLSI-BAM instructions. The instruction set generated by ASIA is identical to the VLSI-BAM instructions in the complete segments. However, this seemly obvious coincidence is the result of a non-trivial design process. Note that `inst5 (swt)` and `inst24 (uni)` are complex instructions, containing multiple, conditional micro-operations, `[ceTe_jpi(1),ceTe_jpi(2)]`,and `[ceTe_mrit,cond2, cond4]`, respectively. For these two instructions, there are too many operands, which are necessary to control the operations, to be fit into the 32 -bit instruction word. The design process tries to split these micro-operations into several time steps, resulting in more instructions. The increased numbers of instructions and time steps are not favored by the objective function. Therefore, the design process then tries to encode the operands in order to reduce the required instruction bits. It takes one and five encoding transformations to satisfy the instruction word

```
label(prolog1).
   cmp(eq,r(0),r(1)).
   bt(fail).
label(prolog2).
   cmp(ne,r(0),r(1)).
   bt(fail).
label(prolog3).
   cmp(lts,r(0),r(1)).
   bt(fail).
label(prolog4).
   cmp(ges,r(0),r(1)).
   bt(fail).
label(prolog5).
   cmp(ltu,r(0),r(1)).
   bt(fail).
label(prolog6).
   cmp(geu,r(0),r(1)).
   bt(fail).
label(prolog7).
   cmp(tageq,r(0),r(1)).
   bt(fail).
label(prolog8).
   cmp(tagne,r(0),r(1)).
   bt(fail).
label(prolog9).
   btgeq(1024, r(0), l(fail)).
label(prolog10).
   btgne(1024, r(0), l(fail)).
label(prolog11).
   ldi(1024,r(16)).
label(prolog12).
   jmpr(r(0)).
label(prolog13).
   rd(pc,r(0)).
label(prolog14).
   ld(r(0),r(1)).
label(prolog15).
   st(r(1),r(2)).
label(prolog16).
   lea(tvar^r(0),r(1)).
label(prolog17).
   addi(r(0),1024,r(1)).
label(prolog18).
   add(r(0),r(1),r(2)).
label(prolog19).
   sub(r(0),r(1),r(2)).
label(prolog20).
   and(r(0),r(1),r(2)).
label(prolog21).
   or(r(0),r(1),r(2)).
label(prolog22).
   xor(r(0),r(1),r(2)).
label(prolog23).
   sra(r(0),r(1),r(2)).
label(prolog24).
   srl(r(0),r(1),r(2)).
label(prolog25).
   sll(r(0),r(1),r(2)).
label(prolog26).
   dref(r(1)).
label(prolog27).
   mov(r(30),r(16)).
   umax(r(27),r(30),r(30)).
label(prolog28).
   swt(r(0),tlst=l(true),tvar=l(fail)).
label(prolog29).
   uni(1024,r(7)).
   bt(fail).
```

Figure 8.3 Complete segments for Prolog (Table 8.3 in [36])

ID	Binary tuple	Register transfers	Equivalent BAM instruction
inst1	`< [ceFe_jpi], [] >`	if (not tag(R_1)=T) { pc ← pc + Immed }	btgne
inst2	`< [ceTe_jpi], [] >`	if (tag(R_1)=T) { pc ← pc + Immed }	btgeq
inst3	`< [ceT_ji], [] >`	if(tf=1) { pc ← Immed }	bt
inst4	`< [jri], [] >`	pc ← R_1+ Immed	jmpr
inst5	`< [ceTe_jpi(1),ceTe_jpi(2)], [[ceTe_jpi(1)-regR, ceTe_jpi(2)-regR]] >`	if (tag(R_1)=T_1) { pc ← pc + Immed_8_1; if (tag(R_1)=T_2) { pc ← pc + Immed_8_2	swt
inst6	`< [ri], [] >`	R_1 ← Immed	ldi
inst7	`< [rmd], [] >`	R_1 ← m(R_2+Immed)	ld
inst8	`< [rr], [] >`	R_1 ← R_2	mov
inst9	`< [rr_pc], [] >`	R_1 ← pc	rd
inst10	`< [rra], [] >`	R_1 ← R_2 + R_3	add
inst11	`< [rrai], [] >`	R_1 ← R_2 + Immed	addi
inst12	`< [rrait], [] >`	R_1 ← T^(R_2 + Immed)	lea
inst13	`< [rran], [] >`	R_1 ← R_2 and. R_3	and
inst14	`< [rreo], [] >`	R_1 ← R_2 xor. R_3	xor
inst15	`< [rro], [] >`	R_1 ← R_2 or. R_3	or
inst16	`< [rrs], [] >`	R_1 ← R_2 - R_3	sub
inst17	`< [sll], [] >`	R_1 ← R_2 << R_3	sll
inst18	`< [sra], [] >`	R_1 ← R_2 >> R_3 (arithmetic right shift)	sra
inst19	`< [srl], [] >`	R_1 ← R_2 >> R_3 (logic right shift)	srl
inst20	`< [cond1], [] >`	tf ← R_2 OP. R_3	cmp
inst21	`< [dref], [] >`	R_1 ← dereference(R_1)	dref
inst22	`< [max], [] >`	R_1 ← max(R_2, R_3)	umax
inst23	`< [mrd], [] >`	m(R_1+Immed) ← R_2	st
inst24	`< [ceTe_mrit,cond2,cond4], [imp(ceTe_mrit-t(tvar)), imp(cond4-t(tvar)), [ceTe_mrit-immed_17, cond2-immed_17], [ceTe_mrit-regR0,ceTe_mrit-regR,cond2-regR,cond4-regR], [ceTe_mrit-tag,cond2-tag]] >`	if (tag(R_1)=tvar) { m(R_1) ← T^Immed}; tf ← R_1 == T^Immed; if (tag(R_1)=tvar) {tf ← 1}	uni

Table 8.14 The instruction set IS_{prolog} for the complete segments in Figure 8.3 on page 130

width constraints, respectively. They are shown in the second arguments of the binary tuples of `inst5` and `inst24`.

8.3.2 Using ASIA for synthesis

At the next step, the complete segments are added to the given application `con3`. The application has one major inner loop body. The number of iteration of the loop body is equal to the length N of the first string. In order to investigate how run time behavior[4] of the loop body affects instruction set design, three versions of the application, namely `con3`, `con25` and `con100`, are created by running the application with the first string of length N=3, N=25 and N=100, respectively. The execution counts of basic blocks for the three versions are manually derived.

The instruction sets are synthesized by ASIA for the modified application. The synthesized instruction sets become complete instruction sets for Prolog that are customized for the application under different input lengths. The synthesis results are shown in Figure 8.4 on page 134. For comparison, the application is also compiled into the VLSI-BAM code with the Aquarius Prolog Compiler [79]. The resource allocation of the VLSI-BAM architecture and the results of the compiled code are listed in the figures as a known design point. Note that the results of the VLSI-BAM architecture are listed in the figures with the lagend ‘BAM’, for the sake of simplicity. When compared to VLSI-BAM’s results, the results of ASIA are listed with the lagend ‘ASIA’.

Figure 8.4 (a) shows the resource allocation. The application `con3` gets the minimal allocation [2R, 1W, 1M, 1A]. As the number N of iteration of the inner loop increases, more resources are allocated to utilize the available MOP parallelism in the loop body ([3R, 1W, 1M, 2A] for N=25, [4R, 2W, 2M, 2A] for N=100). The increased

4. The run time behavior depends on the input patterns (work load).

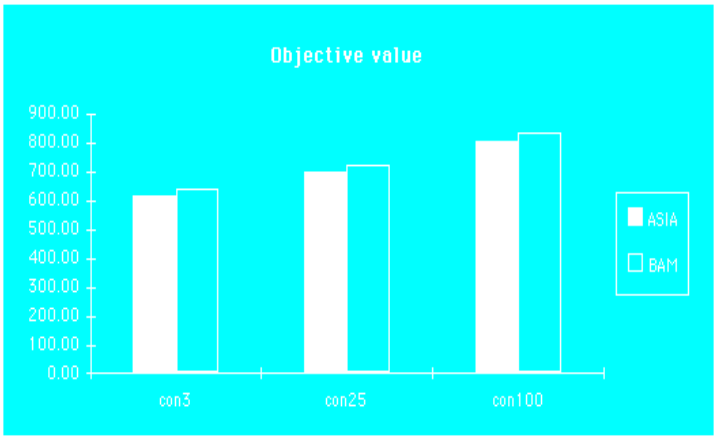
resources are justified by the performance improvement, according to the objective function (to be discussed later). Note that none of these resource allocation cases is equivalent to the resource allocation of VLSI-BAM.

Figure 8.4 (d) shows the sizes of synthesized instruction sets and VLSI-BAM's instruction set. All three synthesized instruction sets contain the base Prolog instructions IS_{prolog} (see Figure 8.14 on page 131), as indicated by the 'Base' lagend in the figure, which is the result of the included complete segments. In addition, the sizes of the synthesized instruction sets are smaller than VLSI-BAM's, while still maintaining general support for Prolog compilation. Note that although the instruction sets of `con25` and `con100` have the same size, many of their instructions are different. Some of `con100`'s instructions are more powerful and consume more hardware resources than `con25`'s. Similarly, some of `con25`'s instructions are more powerful and consume more hardware resources than `con3`'s.

The dynamic cycle counts, which are the cycle counts when executing the assembly code, are shown in Figure 8.4 (b). The synthesized instruction sets produce less dynamic cycle counts than VLSI-BAM's in `con25` and `con100`. The dynamic cycle count of the synthesized instructions for `con3` is about 6% worse than VLSI-BAM's, at the benefit of less hardware resources and instruction set size. The static code sizes are shown in Table 8.15. All synthesized instruction sets produce larger static code (about 7% larger) than VLSI-BAM. This is due to the fact that the static code size is not part of the given objective function such that ASIA does not attempt to minimize it. It will be interesting to conduct further experiments in the future by incorporating the static code size into the objective function.

(a).

(b). (c).



(d). (e).

Figure 8.4 Synthesis results for the applications: `con3`, `con25` and `con100`

Finally, the overall ‘goodness’, indicated by the value of the objective function[5], of the synthesized designs and VLSI-BAM’s design is given in Figure 8.4 (e). The results show that the synthesized designs are better than VLSI-BAM’s for the given application and objective function. However, it is not to conclude that the synthesized designs are customized toward the given application and input patterns, while VLSI-BAM is designed for general purpose and is not balanced toward the objective function in EQ 10.

8.3.3 Using ASIA for design exploration

The experiments conducted so far demonstrate the straightforward use of ASIA: synthesis. The following experiments demonstrate another use of ASIA: design space exploration. For each application, objective functions EQ 11, EQ 12 and EQ 13 are given to ASIA to synthesize instruction sets and allocate resources.

$$\text{Objective} = 2R + 3W + 5M + 4A\text{...(H/W cost)} \qquad \text{EQ 11}$$

$$\text{Objective} = C\text{...(Cycle count)} \qquad \text{EQ 12}$$

$$\text{Objective} = I\text{...(Instruction set size)} \qquad \text{EQ 13}$$

The objective function EQ 11 guides ASIA to minimize the total hardware resources used, assuming the costs of other design metrics (instruction set size, static code size, cycle count) are free. Therefore, this objective function will find the minimal machine in the design space. Similarly, EQ 12 will find the most powerful machine (with least cycle count), at the costs of hardware resources and instruction set size; EQ 13 will find the machine with a minimal instruction set, at the costs of hardware resources and cycle counts. These objective functions are used to explore the boundaries of the multi-dimensional design space.

5. A smaller value indicates a better design.

The experiments results for `con3`, `con25` and `con100` are given in Table 8.15, Table 8.16 and Table 8.17, respectively. In each table, each row lists the result of the experiment with the objective function given in the first column. The results of VLSI-BAM and the designs synthesized in Section 8.3.2 with the global objective function EQ 10 on page 122 are also provided for reference. For global comparison, the results of all experiments are evaluated with the objective function EQ 10, and the obtained values are given in the last column.

Objective Function Used	RF Read Port	RF Write Port	Mem Port	ALU	Inst. set size	Static code size	Cycle count	Obj. Value of EQ 10
H/W cost (EQ 11)	2	1	1	1	42	215	325	636.38
Cycle count (EQ 12)	5	2	3	2	62	169	247	651.94
Inst. set size (EQ 13)	3	1	2	2	32	203	307	631.68
Obj. function (EQ 10)	2	1	1	1	36	194	283	616.54
VLSI-BAM	3	2	2	1	56	184	266	640.35

Table 8.15 Design space exploration for `con3`

Objective Function Used	RF Read Port	RF Write Port	Mem Port	ALU	Inst. set size	Static code size	Cycle count	Obj. Value of EQ 10
H/W cost (EQ 11)	2	1	1	1	42	215	787	724.82
Cycle count (EQ 12)	4	2	2	3	56	168	532	719.66
Inst. set size (EQ 13)	3	1	2	2	32	203	747	720.61
Obj. function (EQ 10)	3	1	1	2	38	197	589	697.84
VLSI-BAM	3	2	2	1	56	184	618	724.65

Table 8.16 Design space exploration for `con25`

Objective Function Used	RF Read Port	RF Write Port	Mem Port	ALU	Inst. set size	Static code size	Cycle count	Obj. Value of EQ 10
H/W cost (EQ 11)	2	1	1	1	42	215	2362	834.73
Cycle count (EQ 12)	4	2	2	3	56	168	1507	823.79
Inst. set size (EQ 13)	3	1	2	2	32	203	2247	830.74
Obj. function (EQ 10)	4	2	2	2	38	195	1540	803.95
VLSI-BAM	3	2	2	1	56	184	1818	832.55

Table 8.17 Design space exploration for `con100`

Several interesting observations can be derived from the results. (1) *Design boundaries*. The design space of these applications are bounded by the following: register-file read ports = 2 ~ 5, register-file write ports = 1 ~ 2, memory ports = 1 ~ 3, ALUs = 1 ~ 3, instruction set size = 32 ~ 56, static code size = 168 ~ 215, cycle counts = 247 ~ 325 (for `con3`), 532 ~ 787 (for `con25`), 1507 ~ 2362 (for `con100`). The designs synthesized by the global objective function EQ 10 on page 122 fall reasonably within the design space. (2) *Performance/cost*. Performance and cost are mostly bounded by the performance (EQ 12) and hardware cost (EQ 11) objective functions. The performance objective function results in a design, called the ‘maximal machine’, with least cycle counts and maximal hardware resources, static code size and instruction set size. On the other hand, the hardware cost objective function results in a design, called the ‘minimal machine’, with minimal hardware resources and largest static code size and cycle count. (3) *Input-pattern invariant objective functions*. The designs produced by the objective functions of hardware resource (EQ 12) and instruction set size (EQ 13) are not sensitive to the input patterns (work load) of the application. The objective function EQ 12 produces the same design for all three input patterns. This is also true for EQ 13. (4) *Correlation between the instruction set size and hardware resources*. It is interesting to note

that minimizing the instruction set size with the objective function EQ 13 does not necessarily minimize the hardware resources. For all three cases, the design with minimal instruction set size consumes modest amount of hardware resources and has modest performance in cycle count and static code size. The minimal instruction set is obtained by enforcing each parallel MOP pattern that happens frequently into a single instruction when splitting the pattern into several instructions will otherwise increase the instruction set size. Packing a frequent MOP pattern into a single instruction requires more hardware resources than the minimal machine does (2R, 1W, 1M, 1A). In addition, these powerful instructions help in reducing the cycle count and static code size. Therefore, the design synthesized by the objective function EQ 13 has smaller cycle count and static code size than the minimal machine.

In summary, the experiments conducted in Section 8.3.2 and Section 8.3.3 suggest that the architectural properties vary with applications and their typical input patterns. In application-specific environments, general purpose instruction set processors might not provide the best performance and cost tradeoff. Instead, instruction sets and hardware resources that are customized toward the dedicated applications may achieve better performance and cost tradeoff. Therefore, tools such as ASIA are necessary to the design of application-specific instruction set processors.

8.3.4 Comparison with other approaches

It is difficult to fairly compare different approaches to automatic instruction set design. These approaches evolve from different research disciplines such as computer architecture, compiler and high level synthesis. They have different concerns, machine models and problem formulations. In addition, many do not publish detailed data in their experiments. Therefore, determining a set of objective metrics for comparing these

approaches is not a easy task. Qualitative comparison, such as the input/out behavior, problem formulation, machine model, has been given in Section 2.3 on page 21, Section 2.6 on page 27 and Section 2.7 on page 29. In this section, quantitative comparison is performed for ASIA and Holmer's work [36] since they are the closest approaches among related work.

- *Algorithm speed*. Holmer's approach generates instructions by executing the simulator of the microarchitecture. The execution of the simulator is one of the major overhead of his approach. ASIA improves this overhead by representing the machine with the microarchitecture specification language in Section 4.2.1 on page 59. The declarative nature of the specification language helps reducing the overhead when combining MOPs into instructions. The algorithm run time of Holmer's algorithm is not published. However, informal experiments have shown that ASIA has a speedup of over seven times over Holmer's for problems of similar size and under the same hardware resource constraints in which hardware resources are given by the designer.[6]

- *Synthesis domain*. The major motivation of developing ASIA is to extend the synthesis domain to cover both instruction set design and hardware resource allocation. As the experiments in Table 8.13 on page 128 shows, this co-synthesis has about three times performance improvement over an iterative approach in which ASIA is used for instruction set design under hardware resource constraints (equivalent to Holmer's approach) and is run several times for several feasible resource configurations. According to previous paragraph, each iteration of ASIA running under hardware resource constraints is about seven times faster. Therefore, the co-synthesis approach

6. The largest experiment conducted by Holmer is the 279 segments in page 160 of [36]. The average size of the segments is 3.29. His algorithm processed the 279*3.29 (918) cycles of instructions with more than one week of computation (according to a private conversation with Holmer). ASIA can be configured to run with the same hardware constraints used by Holmer (as shown in Section 8.1). When used in the same configuration, ASIA processed benchmarks with similar size within 24 hour on the same workstation HP750 (refer to the benchmark *circuit* in Table 8.7 on page 118).

of ASIA is estimated to be about at least 21 (7x3) times faster than Holmer's approach. In addition, ASIA also generates complete assembly code for benchmarks. This approach has the advantage of validating the utilization of the synthesized instruction set. Holmer's approach does not generate complete assembly code. It relies on manual modification of the compiler backend to utilize the new instruction set. This approach creates a gap between the design tool and assembly code: the optimality achieved by the design tool may unnecessarily be reproduced in the final assembly code.[7]

- *Design results*. Both ASIA and Holmer's approach synthesize similar instructions. However, the final instruction sets may vary in some instructions whose utilization rates are low, due to the difference in the search methods when deciding which instructions with lower utilization to be included or excluded. In addition, ASIA may generate instruction sets with different combination of hardware resources which are suitable for the characteristics of corresponding applications and input patterns.

  In terms of cycle counts, both approaches have their limitations in obtaining precise measures. As discussed in the previous paragraph, Holmer's approach relies on manual estimation. First, the Aquarius Prolog Compiler compiles the benchmarks with the subset of the synthesized instruction set which is common to VLSI-BAM instructions. Second, performance improvement due to the synthesized instructions not in the common subset is estimated by manually browsed through the execution profiles. The conclusion is that the synthesized instruction set[8] results in about 10% more cycles than VLSI-BAM. As for ASIA, due to the lack of profiling tools, precise cycle counts can be obtained only for small benchmarks whose profiles can be manually derived.

7. See paragraph 5 in page 167 of [36].

8. using the same data path as VLSI-BAM except no support for double word memory access and three-way branch

According to Figure 8.4 on page 134 (b), the cycle counts (normalized to VLSI-BAM's) of instruction sets synthesized by ASIA for the string concatenation program range from 0.85 (with resources 4R, 2W, 2M, 2A) to 1.06 (with resources 2R, 1W, 1M, 1A). For larger benchmarks, the cycle counts are estimated by assuming that the cycle count is proportional to the static code size. Based on this assumption, the normalized cycle counts of larger benchmarks are about 1.10 which is consists with the results in Figure 8.4 (c). It is expected that the actual cycle counts should be better than what are estimated from static code size since ASIA spends more efforts in optimizing the compilation of frequently executed basic blocks. The cycle count comparison is summarized in Table 8.18.

Instruction Set	Normalized Cycle Count
VLSI-BAM	1.00
Holmer	1.10
ASIA	0.85* ~ 1.06* (1.10†)

Table 8.18 Comparison of cycle counts for VLSI-BAM, Holmer's approach and ASIA

*. These are actual cycle counts obtained from manually annotated execution profiles (see Figure 8.4 (b)). Different cycle counts are the results of various hardware resource allocation (see Figure 8.4 (a)).
†. This number is estimated from the assumption that cycle count is proportional to the static code size.

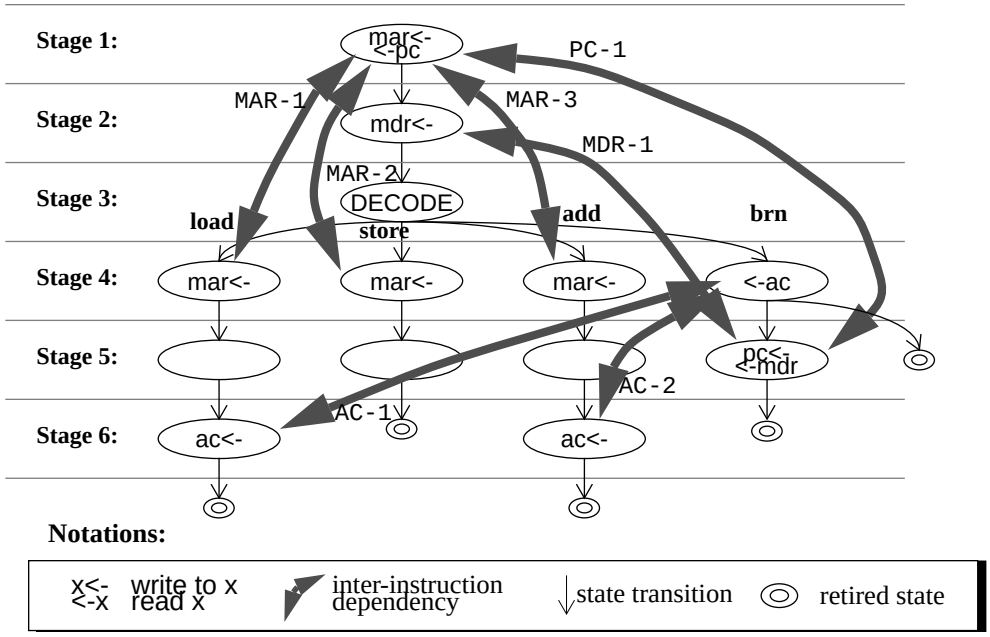
8.4 Synthesis of Microarchitectures and Compiler Backend Interfaces

8.4.1 A small processor with four instructions

This small processor has a minimal achievable latency of five; i.e., any effort to pipeline it with an instruction initiation interval less than five will experience pipeline hazards. We now show how we can pipeline this processor efficiently at the instruction

initiation interval of *one*, for which other pipeline synthesis techniques provide very limited synthesis power.

Figure 8.5 (a) shows the pipeline schedule of this small processor: a simplified finite state representation of the loop body and the pipeline stages. Bubbles in the figure represent states.The state in stage 1 sets the instruction address to `MAR`. The state in stage 2 fetches the instruction. The state in stage 3 is the decode state. The opcode is forwarded to latter stages. States in stage 1, 2, and 3 are active in every clock cycle. For



(a).

```
instruction_pattern_count([load,add],0.14).
instruction_pattern_count([store,add],0.13).
instruction_pattern_count([brn,load],0.01).
instruction_pattern_count([add,store],0.13).
instruction_pattern_count([brn,store],0.13).
```

(b).

Figure 8.5 (a). Pipeline stages and state transitions for instructions: load, store, add, brn
(b). Instruction pair frequency

every clock cycle, there is an active state in stage 4, 5, and 6, respectively. States in stage 4, 5 and 6 are conditionally executed, according to the opcodes in the instruction stream. The contents of bubbles are register accesses we are interested in this discussion. Empty bubbles contain RTLs which are not of interest here. The thick bi-directional arcs are forward/backward inter-instruction dependencies associated with these register accesses. These arcs are labelled as `PC-1`, `MAR-1`, `MAR-2`, etc. There are fourteen dependencies in the figure, such as the forward anti-data dependency of `PC`, backward data dependency of `PC` (shown as the bi-directional arc `PC-1`), etc.

Figure 8.5 (b) shows the instruction pair analysis for a synthetic benchmark which we used to evaluate the various resolution strategies. The first field is the instruction pair: [preceding instruction, succeeding instruction]. The second field is the frequency with which the associated instruction pair exists in the dynamic execution trace. The average instruction level parallelism for this benchmark is about one since most of the instructions access the register `AC`. This observation implies that hardware resolutions may be preferable over software resolutions since there will be very few independent instructions which can be reordered into the `NO-OP` slots.

The hardware/software resolution candidates were generated for each dependency. Some of these candidates are listed in Table 8.19. Note that some reordering constraints can be covered by others. For example, reordering constraints derived for the instruction pair (`allInstructions, brn`) from dependency `MDR-1` can be covered by the ones derived from `PC-1`. Two hardware resolution candidates are available: a duplicate register for `mar` (resolving `MAR-1:f` and `MDR-1:b`) and forward registers for `mdr` (resolving `MDR-1:f`). The weights assigned to these hardware resolution candidates by the benchmark analysis is 2.46 and 0.41, respectively, implying that the former hardware resolution candidate is much more effective than the latter.

Dependent Pair: (f=forward, b=backward)	Software Resolution (reorder constraint) <direction, proceeding inst., succeeding inst, reorder distance>	Hardware Resolution <type,target register, number,weight> (d=duplicate register, f=forward register)
PC-1: f	<up, all instructions, brn, 3>	
pc-1: b	<down, brn, all instructions, 4>	
AC-1: f	<up, brn, load, 1>	
AC-1: b	<down, load, brn, 2>	
MAR-1: f	<up, all instructions, load, 2>	<d, mar, 1, 2.46>
MAR-1: b	<down, load, all instructions, 3>	<d, mar, 1, 2.46>
MDR-1: f	<up, all instructions, brn, 2>	<f, mdr, 2, 0.41>
MDR-1: b	<down, brn, all instructions, 2>	

Table 8.19 Hardware/Software resolution candidates

The column ‘Design 1’ of Table 8.20 lists the design with software-only resolutions. PIPER found four down-reordering constraints. The maximal, minimal, and average reorder distances are 4, 3, and 3.25, respectively. This design has an estimated speedup of 2.13 (w.r.t. non-pipelined design).

We now add a duplicate register for `mar` to resolve its output dependencies between stage one and three which involves instructions ‘add’, ‘load’ and ‘store’. These patterns exist in our benchmark with a total frequency of 40%. This hardware resolution removes one down-reordering constraint, improves the performance by 78% (2.13 to 3.80 in speedup) and reduces the reordering complexity by 24.8% (1.33 to 1 in relative time complexity). This is shown in column ‘Design 2’ of Table 8.20. The speedup improvement is very significant since for every ‘add’, ‘load’, and ‘store’ in the execution trace, one ‘nop’ has to be inserted to avoid the output conflict of `mar`, if it were not duplicated. (`mar` is a special register. A duplicate of it requires a two-port memory support).

The column ‘Design 3’ shows the design with the hardware resolution of forward registers. This hardware resolution resolves the forward data dependency `MDR-1`. The reordering constraints by dependency `MDR-1` require that two ‘nop’ be inserted after ‘brn’. A forwarding register chain consisting of two registers can be allocated in stage three and four such that ‘brn’ can carry its own copy of `mdr` along the pipeline until it reaches stage five where it accesses `mdr`, leaving stage two immediately available for next instruction. However, this strategy does not introduce the anticipated performance improvement. None of the reordering constraint is eliminated. The reason is that the reorder distance for the `brn-allInstructions` pairs is dominated by the dependency `PC-1`, instead of `MDR-1` that the forward registers are to resolve.

The column ‘Design 4’ is the design with both hardware resolutions. The result shows that this design has the same performance and reordering constraints as ‘Design 2’. Therefore, the forward registers are wasted resources. The situations of ‘Design 3’ and ‘Design 4’ illustrate the interaction between the resolutions for different dependencies associated with the same instruction pair, and expose the limitation of the simple weighting model described in Section 7.1.2 on page 102: it is effective in sorting the importance of hardware candidates; however, it is insufficient in estimating the exact performance impacts of the hardware candidates, especially for designs with a combination of many hardware resolutions.

One observation about the reordering constraints for various designs of this small processor is that the maximal reorder distance remains as four while the number of constraints decreases slightly as the hardware resource is invested. This is because that the maximal reorder distance in this example is derived from the backward data dependency `PC-1` between stage one and five which does not have applicable hardware resolution.

Most of the pipeline synthesis techniques described in Section 2.4 on page 22 are not suitable in synthesizing circuits with pipeline hazards, except ASPD. For this small

processor, with the synthesis goal of keeping instruction initiation interval as one, ASPD would produce a pipelined design that flushes the pipe for 3, 3, 3, 5 cycles as soon as the instruction ‘load’, ‘store’, ‘add’, and ‘brn’ are decoded, respectively. This design decision actually slows down the pipeline significantly to speedup (w.r.t. to non-pipelined case) of less than 1.5, much less than our case (speedup=3.80 for the ‘Design 2’ in Table 8.20). However, ASPD does not generate any compilation information for the compiler back-end, which simplifies the hardware/software complication.

	Design 1	Design 2	Design 3	Design 4	ASPD’s
Registers (h/w resolution) <type, target register, number> dg = duplicate register fg = forward register		<dg,mar,1>	<fg,mdr,2>	<dg,mar,1> <fg,mdr,2>	n/a
Down-reordering constraint (s/w resolution) <# of constraint, maximal reorder distance,minimal reorder distance, avg. reorder dist.>	<4, 4, 3, 3.25>	<3, 4, 2, 2.67>	<4, 4, 3, 3.25>	<3, 4, 2, 2.67>	n/a
Estimated speedup of the given benchmark (w.r.t. non-pipelined)	2.13	3.80	2.13	3.80	<1.5
Relative time complexity of the reorderer (compiler back-end)	1.33	1	1.33	1	0

Table 8.20 Designs with various combinations of hardware/software resolutions

8.4.2 SM2a processor

The SM2a processor is a 39-instruction micro-processor with both general and special purpose registers. It was used by the Advanced Computer Architecture Laboratory at University of Southern California for the studies of Prolog compilation and design automation. The minimal achievable latency of this processor is five; i.e., any effort to synthesize this processor with an instruction initiation latency less than five will introduce pipeline hazards.

The results of the pipeline hazard resolutions for heavily pipelined SM2a (latency= 1, 2, 3, 4) are presented in Table 8.21. Designs made by MAL-based techniques and ASPD are listed in the table for comparison. Since we were not able to get access to these tools, the data produced for these tools were manually derived based on our best knowledge of their algorithms. The third row is the latency. The fourth row lists the number of dependencies to be dealt with for each instruction initiation latency considered. This number implies the 'difficulty' of the problem. The fifth row lists the number of hardware resolution candidates for duplicate registers, and forward registers, respectively. The sixth row lists the hardware resolutions selected by the designer. For example, in design #8, the designer selected one duplicate register and one forward register resolution (1D,1F) out of the possible hardware candidates (1D,4F). The entry with 'no' means that no hardware is selected. The seventh through tenth rows summarize the software resolutions with respect to the design decision made in the sixth row. The eleventh row is the estimated speedup with respect to the non-pipelined case. The last row is the relative time complexity of the reorderer. The experiment took 25 seconds on a HP750 workstation.

Instruction initiation latency	4	3		2		1			1
# of inter-instruction dependencies	54	90		342		484			0
possible hardware resolutions (D = duplicate register, F = forward register)	-	1D		1D,2F		1D,4F			-
selected hardware resolutions	no	no	1D	no	1D	no	1D	1D,1F	1D,3F	-
# of down-reordering	12	23	15	113	111	27	115	113	115	0
max. reorder distance	1	1	1	2	2	4	4	4	4	0
min. reorder distance	1	1	1	1	1	1	1	1	1	0
average reorder distance	1.00	1.00	1.00	1.10	1.10	3.20	2.20	2.22	2.20	0
Design ID#	1	2	3	4	5	6	7	8	9	10*

Table 8.21 Results of pipeline hazard resolution for the sm2a processor

*. Synthesized by ASPD [7]

Several comments can be drawn from the results. First, our techniques outperform other techniques. The maximal speedup (for the given application benchmark) of pipelined SM2a synthesized with MAL-oriented pipeline techniques is 1.2 (design #11, with instruction initiation latency of five). The design #10 synthesized by ASPD can execute at the latency of one; however, for a large portion of the execution time the design flushes the pipeline for both necessary and unnecessary cases. Therefore, it achieved an overall speedup of 1.5. On the other hand, we were able to increase the speedup to 2.37 for the given benchmark, and 6 for benchmarks that have no inter-instruction dependency in the design #8 (instruction initiation latnecy=1, with one duplicate and one forward register as the hardware resolution). The improvement is achieved at the cost of instruction reordering. The compiler backend has to examine 113 pairs of down-reorder constraints.

Second, we were able to generate several design alternatives consisting of different hardware/software combinations. The designs #1~#9 provide choices of instruction initiation latencies from 1 to 4, speedups from 1.1 to 2.37, and relative compiler backend (reorderer) time complexities from 1 to 9.58. The designer can select an appropriate design based on the application environment. For example, for embedded applications such as the cruise control of vehicles, the software is compiled only once and then loaded into the system. Therefore, the designs with higher reorderer time complexity are justifiable. On the other hand, if the designs are to be used as programming tools where compilation happen very often, then the designs with lower reorderer time complexity are more feasible.

Third, The number of dependencies to be resolved grows fast when the instruction initiation latency is decreased. This is because that the higher the degree of pipelin-

ing, the larger the number of pipeline stages; the larger the number of pipeline stages, the more interactions between stages.

Fourth, adopting more hardware resolutions does not necessarily reduce the number of reordering constraints. One of the reasons is that the adopted hardware may deleted a constraint of general cases such as `ro(load,allInstructions,5)` which originally covers `ro(load,add,3)` and `ro(load,sub,4)`. By deleting the general case, special cases such as the latter two will get exposed to the final software resolutions.

Fifth, for each set of hardware resolution candidates, there exists a minimal subset which provides the maximal performance gain such as the subset (1D,1F) of the candidate set (1D,4F) of latency=1. Currently this subset is empirically identified through estimation and experiment. It appears to us that a systematic search for such subset is an interesting future research direction.

Figure 8.6 shows the performance/cost metrics for candidate designs #1 to #10. Figure 8.6 (a) is the average speedup of the given benchmark (w. r. t. non-pipelined

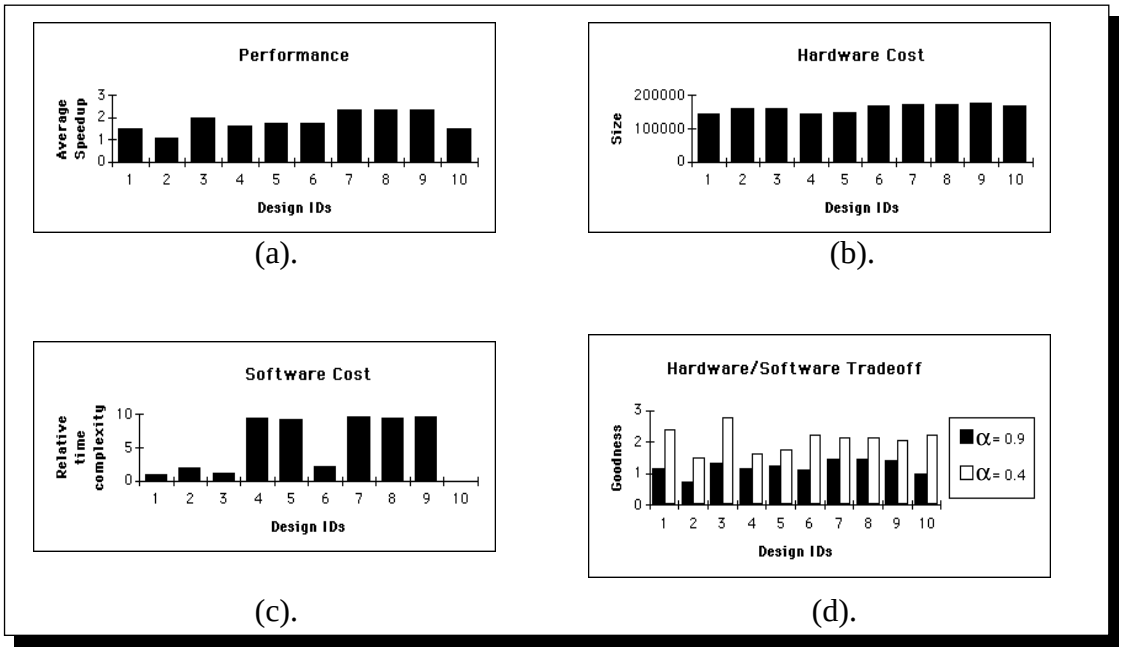


Figure 8.6 Performance/cost tradeoff analysis of SM2a

design). Designs #7, #8 and #9 have the largest speedup. Figure 8.6 (b) shows the hardware sizes of candidate designs. The hardware size, including both data and control paths, was estimated at the RTL level. We were not able to obtain the actual size of design #10 directly. However, it should have roughly about the same data path size as design #6's since they both have no hardware resolution. The control path of design #10 may be more complicated than that of design #6, because the former has to flush the pipeline. As for the cost of software, the time complexities of design #1, #2, #3 and #6 are much smaller than others', due to smaller number of reorder constraints, as shown in Figure 8.6 (c). Note that design #10 does not require reordering, thus it has zero time complexity in the figure.

Based on the performance and cost of hardware and software in Figure 8.6 (a) through (c), the global hardware/software tradeoff analysis was conducted for two cases: α=0.9 and α=0.4 in EQ 8 on page 109, as shown in Figure 8.6 (d). The case α=0.9 assigns more emphasis on the hardware cost and tolerates more software cost. The design with the best "performance/cost" ratio in this case is design #7 (with instruction initiation latency=1, one duplicate register, and 312 reorder constraints). On the other hand, the case α=0.4 assigns less importance to the hardware cost, and is more sensitive to the software cost. In this case, the best design is design #3 (with instruction initiation latency=3, one duplicate register, and 15 reorder constraints).

In summary, we have shown that our method is able to improve the throughput of the pipelined SM2a. The performance is much better than that of pipeline synthesis techniques currently available. Besides, our method provides better flexibility in producing designs for different application domains. The designers can customize the designs either for embedded systems where most of the applications are compiled beforehand, thus making software cost (compilation time) more affordable as shown in our α=0.9 case, or for application development systems where lots of applications are constantly

compiled during the life time of the machines, thus making software cost less affordable as shown in our α=0.4 case.

8.4.3 The TDY-43 processor

The TDY-43 processor was designed about twenty years ago, and was used for aviation control in helicopters [78]. It has 256 instructions supporting fix-point, fractional, and two's complement operations on nine registers, a wide variety of addressing modes, and some external I/O controls. It was built on six boards. While it is still widely in service, its parts become obsolete and raise a difficult maintenance problem. Therefore, a customized single-chip re-implementation is desirable. The ADAS design automation system [69] was used to generate a gate-array implementation from the instruction set specification. PIPER is the high level synthesis tool of ADAS, and was used to explore possible pipeline implementations of TDY-43.

Instruction initiation latency	8	6	4		3		2		1
# of inter-instruction dependencies	3968	5167	7924		16307		47091		105654
possible hardware resolutions (D = duplicate register, F = forward register)	2D, 1F	3D, 2F	4D, 7F		3D, 8F		4D, 19F		n/a
selected hardware resolutions	no	no	no	3D, 2F	no	3D, 7F	no	4D, 19F	n/a
# of down-reordering	2180	2199	2406	2319	5245	5191	2814	11991	n/a
max. reorder distance	3	4	6	6	8	8	13	13	n/a
min. reorder distance	1	1	1	1	1	1	1	1	n/a
average reorder distance	2.14	2.93	4.36	4.48	3.24	3.27	7.95	2.62	n/a

Table 8.22 Results of pipeline hazard resolution for the TDY-43 processor

Table 8.22 shows some results of TDY-43. The design with latency of one was not completed since PIPER could not finish within a reasonable time (more than two weeks). The application benchmark was not available to us at the time of experiment.

Therefore, we do not show the estimated performance with respect to the benchmark. According to the experiment, TDY-43 can be pipelined at the latency of two, with 2814 reorder constraints. The large number of reorder constraints is primarily due to complex instructions such as normalization of fractional numbers, multiplication/division, and shifting by an arbitrary amount (0~64 bits). Even at the latency of eight, there are still 2180 reordering constraints required. At this latency, most of the simple instructions can be completed within a single stage. The hardware/software resolutions are mainly for resolving the hazards for the instruction pairs involving those complex instructions. Judging from this experiment as well as the fact that the company that designed TDY-43 would have difficulty in re-compiling their applications, we concluded that pipelining with software support (reordering) was not a feasible design style for TDY-43, instead, a sequential implementation was recommended.

Chapter 9 Conclusions

9.1 Summary

The design of instruction set processors includes several interdependent subtasks such as instruction set design, microarchitecture design and code generation. These design subtasks have been mostly treated as independent ones. The interdependency is dealt with iteration between these subtasks. The iteration is usually conducted in informal ways, much depending on the experience and ingenuity of the designer. In order to control the tradeoffs between competing design options, application benchmarks which represent the most typical computation to be executed on the target processor are given to the designer for performance and cost analysis.

This dissertation presents systematical approaches to the design of instruction set processors at the architectural and microarchitectural levels. Subtasks are solved in an integrated way.

At the architectural level, a set of techniques have been developed to solve a combined problem of instruction set design, hardware resource estimation and code generation. The approach takes as input the application benchmarks, architectural template, objective function and design constraints, and generates as output the application-specific instruction set, resource allocation (which instantiates the architecture template) and assembly code for the application benchmarks. The approach is based on an integrated problem formulation: a simultaneous scheduling/allocation problem with an integrated instruction formation process.

A specification language for the architecture template is proposed. A preprocessor then maps the application to dependency graphs consisting of MOPs that are supported by the architecture template. The MOPs are scheduled into time steps, subject to

constraints. During the scheduling process, instructions are formed and resources are allocated at the same time. A binary tuple is used to describe the semantics of instructions. The binary tuple is the key idea which links the instruction formation to the scheduling process. An efficient simulated annealing algorithm and a set of move operators are presented to manipulate the design state and solve the schedule. The method has been implemented in our design automation system ASIA (Automatic Synthesis of Instruction-set Architecture) which is the architectural domain tool of the ADAS (Advanced Design Automation System) full-range design automation system for instruction set processors.

At the microarchitectural level, a set of techniques have been developed to solve the combined problem of pipelined microarchitecture design and code optimization. The approach takes as input the instruction set architecture specification and application benchmarks, and generates as output the RTL implementation of the pipelined microarchitecture and the reordering table which serves as an interface to the reorderer of the compiler backend for the target processor. The unique feature which differentiates the approach from other microarchitectural (behavioral) synthesis approaches for pipelined instruction set processors is its problem formulation based on hardware/software concurrent engineering. The key of the approach is the capability of systematically analyzing potential pipeline hazards and applying appropriate resolutions.

An extended taxonomy of inter-instruction dependencies is proposed for the analysis of register-related pipeline hazards in instruction set processors. Hardware and software solution for the resolution of pipeline hazards are developed. These solutions include forwarding/duplicate registers in hardware, and up/down instruction reordering in software (compiler back-end). The register-related pipeline hazards are resolved according to the types of the inter-instruction dependencies they involve. In an application specific environment, the combination of hardware and software solutions can be

tuned towards the characteristics of the application benchmarks. The design procedure for pipeline hazard resolution and the related algorithms have been described. The set of techniques have been implemented in the PIPER design automation system, which is the microarchitectural (behavioral) domain of the ADAS system.

PIPER can be used in two ways. First, it serves as a post-processor for ASIA by generating the reordering table for the reorderer and the RTL implementation of the microarchitecture for the instruction set synthesized by ASIA. The detailed information of the microarchitecture implementation can be fed back to ASIA to refine the design process by adjusting the performance and cost estimation. Second, PIPER can be used without ASIA as a hardware/software co-design tool to explore the design space of pipeline structure and software (reorderer) complexity.

9.2 Contributions

The work in this dissertation provides integrated problem formulations to the problems of instruction set design, microarchitecture design and code generation, which help better understanding of their design processes and interactions. Based on the formulations, efficient algorithms are developed to make the design automation systems practical in synthesizing designs and exploring design tradeoffs among competing factors in both hardware and software.

At the architectural level, ASIA can be used as a fast prototyping tool to determine feasible initial designs for new instruction set architecture design, and as a tool for the evaluation of existing instruction set architectures in application specific environment. Some case studies of ASIA's applications have been presented in Section 8.1 on page 112 and Section 8.2 on page 122.

PIPER can be used to generate the register transfer level (RTL) implementation for the architecture synthesized by ASIA, providing more accurate performance/cost

measures which can be fed back to ASIA to guide the design process. In addition, PIPER can be used to explore the tradeoffs between microarchitecture and the complexity of the compiler backend. Furthermore, it can be used to extend the service life and improve the performance of existing instruction set architecture by simultaneously re-implementing hardware with pipeline structures and generating the interface to patch the original software environment so as to take advantage of the pipelined microarchitecture. Examples of PIPER's applications have been given in Section 8.4 on page 141, including the synthesis of an industrial processor TDY-43.

9.3 Future Directions

The techniques presented in this dissertation serve as a basic framework for investigating hardware, software and interface problems of instruction set processor design. Further improvements and enhancements can be made to expand the applicability and practicality of the techniques.

- *Implementation limit*s. The current implementation of the techniques are subject to certain limits which can be improved in the future. (1) There are some interesting architectural properties which have not been considered in ASIA. For example, multiple-word memory access with a single address is a way to increase memory data bandwidth while consuming limited addressing bandwidth. This feature has been proved as an effective way to increase performance of Prolog applications [35]. (2) The complication between data and control dependencies have not been considered yet. For example, the value of the program counter may be stored into a general purpose register for further data manipulation, e.g. addition. This operation converts a control data into a computational data, which turns a control-related dependency into a data-related dependency. When move related micro-operations around, the value of the data, derived from the program counter, may have to be adjusted. (3) The probabi-

listic approach of ASIA is inefficient in locating and pruning inferior instructions which happens very rarely in the application. A post phase after the simulated annealing process would be very helpful to refine the results. (4) Although the proposed techniques in microarchitecture synthesis are general enough to support the use of register files and multi-cycle non-pipelined operations in data path, the implementation of PIPER is not ready for these features. This is in part due to the original assumption of the ADAS system that registers are to be individually named, and in part due to the tight schedule for developing PIPER. The lack of support for register files causes some problems in integrating PIPER with ASIA since the instruction sets synthesized by ASIA may contain register files.

- *Algorithmic limits*. The simulated annealing approach used in ASIA is a convenient and flexible way for fast prototyping. It also offers the opportunity of incremental design improvement since it works by applying operators to improve a known solution. However, it is often difficult to control the quality and may be subject to the convergence problem. Based on the lessons learned from the current approach, it is possible to develop algorithmic approaches that maintain the fore mentioned advantages, while providing faster speed and solving the disadvantages. One solution is to develop a post phase after the simulated annealing process to perform local transformations, in order to improve the design quality. However, this solution does not provide speed up in design time. Innovative approaches are necessary to provide significant speed up in design time.

- *Realistic application benchmarks*. Although the current approach of ASIA has achieved a significant speed improvement over Holmer's approach, further improvements are necessary to deal with realistic application benchmarks such as operating system kernels, compilers, simulators, CAD and DSP applications. A straight forward solution is to select a representative portion from the benchmarks and apply the cur-

rent approach to synthesize the application-specific instruction set, generate assembly code and estimate the resource allocation. A code generation process can be used to compile the unselected portion. A problem caused by this solution is how to select the representative portion, which is to be investigated. In addition, this solution relies heavily on the availability of an effective retargetable code generator.

- *Global optimization*. The current approach of ASIA uses basic blocks as boundaries for micro-operation movement, which is subject to the limited parallelism available within basic blocks. Global optimization can be performed on the application benchmarks to extract more parallelism before they are given to ASIA. Some forms of global optimization, such as branch prediction and speculative execution, may require dedicated architectural or microarchitectural features to be incorporated into the machine model, which further complicates the performance/cost tradeoffs between hardware and software.
- *Architectural and microarchitectural enhancement*. (1) Several architectural or microarchitectural properties have not been considered in the current implementation, such as register allocation, the size of the register file and the size of immediate data. These are practical design issues which can not be ignored. However, these issues have a lot of impact on the performance and cost of both hardware and software, which makes it difficult to handle these issues properly. (2) Cache miss and interrupt/trap are important activities in modern processors. These activities can be made either explicit to the design system by describing them in the application benchmarks or the instruction set architecture specification, or implicit to the design system by treating them as built-in modules which are automatically included to the target processor by the design system. For the implicit case, analytical models are necessary to predict the performance influence of these built-in modules. (3) Extensions can be made to

expand beyond uniprocessors. Some interesting features include internal opcode control model, VLIW (very long instruction word) architecture, superscalar architecture, etc.

- *Exploit of architectural and microarchitectural properties.* As what have been demonstrated by the experiments in Chapter 8, many architectural and microarchitectural properties can be exploited with our techniques, including instruction word width, instruction set size, instruction set variation, performance/cost variation, compilation complexity and classification of application benchmarks. However, in the experiments, many of the properties were manually analyzed, based on the synthesis results. The manual approach limits the scopes of exploitation. Further analysis tools are required to automate the analysis processes.

- *Software development environment.* For every instruction set processor developed, there is a set of software required to test, verify and run the processor, such as simulators and compilers. Software related to the architecture and microarchitecture design of processors includes instruction-level simulators, microarchitecture simulators, code generators, reorderers[1], peephole optimizers, etc. It is desirable to generate these software tools automatically when designing new processors since software development usually takes equal or more time than hardware development [66].

- *Redesign based on feedback information.* One of the advantages of integrated design automation systems is that design information generated by one tool can be easily used by other tools to improve design quality. One style, which is specifically suitable on our ADAS framework, of using the design information is feedback-oriented design (redesign). Design details obtained from lower level design tools can be fed back to higher levels to modify design heuristics and estimation. Through modification and

1. The reorderer can be embedded in the peephole optimizer.

redesign, design with better quality or closer to designer's intention can be obtained. On the contrary to the design paradigm that is based on complicated algorithms in order to generate a good design within a single design iteration, a feedback-oriented design paradigm relies on many iterations of simple, easy-to-control algorithms to obtain a good design. It is believed that a feedback-oriented design paradigm is suitable for our architectural-level and microarchitectural-level design. Because it is difficult to estimate performance and cost accurately at such levels without lower level information, the feedback-oriented approach becomes more practical for our design problems. To construct a feedback-oriented design paradigm requires that information flow and user interfaces of tools be clearly defined. In addition, tools that separate their design engines from design rules which are design sensitive will facilitate performance results from the exploitation of feedback of more concrete design levels.

References

[1] Alauddin Alomary, et al., “PEAS-I: A Hardware/Software Co-design System for ASIPs,” *Proc. of EURO-DAC*, 1993

[2] Alauddin Alomary, et al., “An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint,” *Proc. of the International Conference on Computer-Aided Designs*, Nov. 1993

[3] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Adison-Wesley, 1985

[4] Pradip Bose and Edward S. Davidson, “Design of Instruction Set Architectures for Support of High-Level Languages,” *Proc. of the 11th Annual International Symposium on Computer Architecture*, 1984

[5] J. P. Bennett, *Automated Design of an Instruction Set for BCPL*, Technical Report 93, University of Cambridge, Computer Laboratory, 1986

[6] J. P. Bennett, *A Methodology for Automated Design of Computer Instruction Sets*, Ph.D. thesis, University of Cambridge, Computer Laboratory, 1988. Also available as Technical Report 129

[7] Mauricio Breternitz Jr. and John Paul Shen, “Architecture Synthesis of High-Performance Application-Specific Processors”, *Proceedings Design Automation Conference*, 1990

[8] Bill Bush, et al., *The Berkeley Abstract Machine Instruction Manual*, Internal Technical Report, Advanced Computer Architecture Laboratory, University of Southern California, 1990

[9] William Bush et al., “An Advanced Silicon Compiler in Prolog,” *Proceedings International Conference on Computer Aided Design*, 1987

[10] R. G. G. Cattell, “Automatic Derivation of Code Generators from Machine Description,” *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, April 1980

[11] Mike Carlton, Source codes of the Aquarius Prolog Compiler (Back-end), University of California, Berkeley, 1991

[12] Albert E. Casavant, “MIST-A Design Aid for Programmable Pipelined Processors,” *Proc. of the 31st Design Automation Conference*, June 1994

[13] Wei-Kai Cheng and Youn-Long Lin, “Code Generation for a DSP Processor,” to appear in *Proc. of Int’l Symposium on High Level Synthesis,* May 1994

[14] Gino Cheng et al., “Advanced Design Automation System for Microprocessors,”

[15] Richard Cloutier and Donald Thomas, “Synthesis of Pipelined Instruction Set Processors,” *Proc. of the 30th Design Automation Conference,* 1993

[16] M. Corazao, et al., “Instruction Set Mapping for Performance Optimization,” *Proc. of ICCAD,* Nov. 1993

[17] H. J. Curnow and B. A. Wichmann, “A Synthetic Benchmark,” *The Computer J.,* 19:1, 1976

[18] Subrata Dasgupta, *Computer Architecture A Modern Synthesis,* Volume 1, John Wiley & Sons, 1989

[19] Subrata Dasgupta, *Computer Architecture A Modern Synthesis,* Volume 2, John Wiley & Sons, 1989

[20] Subrata Dasgupta, *Design Theory and Computer Science,* Cambridge University Press, 1991

[21] Alvin Despain et al., “Aquarius,” *Computer Architecture News,* March, 1987

[22] Alvin Despain, “The Design System (ASP) of the Aquarius Project,” *Proceedings of the Second International Workshop on VLSI Design,* December, 1988

[23] Srinivas Devadas and Richard Newton, “Algorithms for Hardware Allocation in Data Path Synthesis,” *IEEE Trans. on Computer-Aided Design,* Vol. 8, No. 7, July 1989

[24] J. A. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction,” *IEEE Trans. on Computers,* Vol. 30, No. 7, 1981

[25] D. Gifford and A. Spector, “Case Study: IMB’s system360-370 Architecture,”, *Communications of the ACM,* 30(4):292-307, April 1987

[26] Robert Giegerich, “A Formal Framework for the Derivation of Machine-Specific Optimizers,” *ACM Transactions on Programming Languages and Systems,* Vol. 25 No. 3, July 1983

[27] Gert Goossens, Jan Rabaey, Joos Vandewalle and Hugo De Man, “An Efficient Microcode Compiler for Application Specific DSP Processors,” *IEEE Trans. on Computer-Aided Design,* Vol. 9, No. 9, September 1990

[28] S. L. Graham, “Table-Driven Code Generation,” *IEEE Computer,* August 1980

[29] F. M. Haney, “ISDS-A program that designs computer instruction sets,” *Fall Joint Computer Conference*, 1969

[30] R. Haygood, *A Prolog Benchmark Suite for Aquarius*, Technical Report, UCB/CSD 89/509, University of California, Berkeley, 1989

[31] John Hennessy and Thomas Gross, “Postpass Code Optimization of Pipeline Constraints,” *ACM Tran. on Programming Languages and Systems*, July 1983, pp. 422-448

[32] John L. Hennessy and David A. Patterson, “Computer Architecture A Quantitative Approach,” *Morgan Kaufmann Publishers*, 1990, pp. 257-278

[33] John Hennessy, Norman Jouppi, Steven Prezybylski, Christopher Rowen, and Thomas Gross, “Design of a High Performance VLSI Processor,” *Third Caltech Conference on VLSI*, page 33, 1983

[34] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann publishers, 1990

[35] Bruce Holmer, et al., “Fast Prolog with an Extended General Purpose Architecture,” *Proc. of 27th International Symposium on Computer Architecture*, 1990

[36] Bruce Holmer, *Automatic Design of Computer Instruction Sets*, Ph.D. thesis, Univ. of California, Berkeley, 1992

[37] Bruce Holmer and Alvin Despain, “Viewing Instruction Set Design as an Optimization Problem,” *Proc. of Micro-24*, 1991

[38] Bruce Holmer and Barry Pangrle, “Hardware/Software Codesign Using Automated Instruction Set Design & Processor Synthesis,” *Proc. of Hardware/Software Codesign Workshop*, 1993

[39] Ing-Jer Huang and Alvin Despain, “High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers,” *Proc. of the 29th DAC*, 1992

[40] Ing-Jer Huang and Alvin Despain, “An Extended Classification of Inter-instruction Dependency and Its Application in Automatic Synthesis of Pipelined Processors,” *Proc. of the 26th International Symposium on Microarchitectures*, Nov. 1993

[41] Ing-Jer Huang and Alvin Despain, “Hardware/Software Resolution of Pipeline Hazards in Instruction Set Processors,” *Proc. of the International Conference on Computer-Aided Design,* Nov. 1993

[42] Ing-Jer Huang, Bruce Holmer and Alvin Despain, “ASIA: Automatic Synthesis of Instruction-set Architectures,” *Proc. of SASIMI Workshop*, Nara, Japan, Oct. 1993

[43] Ing-Jer Huang and Alvin Despain, “Synthesis of Application Specific Instruction Sets,” accepted for the *IEEE Trans. on CAD*, 1994

[44] Ing-Jer Huang and Alvin Despain, “Synthesis of Instruction Sets for Pipelined Microprocessors,” *Proc. of the 31st Design Automation Conference*, June 1994

[45] Ing-Jer Huang and Alvin Despain, “Generating Instruction Sets and Microarchitectures from Applications,” accepted to *the International Conference on Computer-Aided Design*, April 1994

[46] Cheng-Tsung Hwang et al., “Scheduling for Functional Pipelining and Loop Winding*”, Proc. 28th DAC,* 1991

[47] Masaharu Imai, Alauddin Alomary et al., “An Integer Programming Approach to Instruction Implementation Method Selection Problem,” *Proc. of Euro-DAC*, 1992

[48] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991

[49] Gerry Kane, *MIPS RISC Architecture,* Prentice-Hall, 1989

[50] Hironobu Kitabatake and Katsuhiko Shirai, “Functional Design of a Special Purpose Processor Based on High Level Specification Description,” *IEICE Trans. Fundamentals,* Vol. E75-A, No. 10, Oct. 1992

[51] Peter M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Company, 1981

[52] Vipin Kumar, “Algorithms for Constraint Satisfaction Problems: A Survey,” *AI Magazine*, spring, 1992

[53] Anshul Kumar and Shashi Kumar, “Automatic Synthesis of Microprogrammed Control Units from Behavior Descriptions,” *Proc. of 26th DAC*, 1989

[54] Tsing-Fa Lee, et al., “An Effective Methodology for Functional Pipelining,” *Proc. of ICCAD*, 1992

[55] Thomas Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, England, 1990

[56] Clifford Liem, Trevor May, Pierre Paulin, “Instruction-Set Matching and Selection for DSP and ASIP Code Generation,” *Proc. of EDAC*, 1994

[57] Shi-Zheng Lin, Cheng-Tsung Hwang and Yu-Chin Hsu, “Efficient Microcode Arrangement and Controller Synthesis for Application Specific Integrated Circuits,” *Proc. of ICCAD*, 1991

[58] F. M. McMahon, “The Livermore FORTRAN Kernels: A Computer Test of Numerical Performance Range,” Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore, December 1986

[59] Michael C. McFarland, Alice C. Parker and Raul Camponsano, “The High-Level Synthesis of Digital Systems,” *Proc. of the IEEE, Vol. 78, No. 2*, February 1990

[60] Glenford J. Myers, *Advances in Computer Architecture*, John Wiley & Sons, 1982

[61] Pierre G. Paulin and John P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s,” *IEEE Trans. on Computer-Aided Design*, vol. 8, No. 6, June 1989

[62] Nohbyung Park and Alice C. Parker, “Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications,” *Trans. on CAD, Vol 7. No. 3*, March 1988

[63] Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982

[64] Nohbyung Park, Rajiv Jain and Alice C. Parker, “Data Path Synthesis of Pipelined Designs: Theoretical Foundations,” in *Progress in Computer-Aided VLSI Design*, Vol. 3, pages 119-156, Ablex Publishing, 1990

[65] J. H. Patel and E. S. Davidson, “Improving the Throughput of a Pipeline by Insertion of Delay,” *IEEE/ACM 3rd Ann. Symp. Computer Arch.*, IEEE No. 76CH0143-5C, 1976

[66] Pierre Paulin, Clifford Liem, Trevor May, Shailesh Sutarwala, “DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective,” to appear in *Journal of VLSI Signal Processing*, 1994

[67] J. Pendleton, et. al., “A 32-bit Microprocessor for Smalltalk,” *IEEE Journal of Solid State Circuits*, SC-21(5):741-749, October 1986

[68] Johan Van Praet, Gert Goossens, Dirk Lanneer, Hugo De Man, “Instruction Set Definition and Instruction Selection for ASIPs,” to appear in *Proc. of Int’l Symposium on High Level Synthesis*, May 1994

[69] Iksoo Pyo, et al., “Application-Driven Design Automation for Microprocessor Design,” *Proc. of the 29th DAC*, 1992

[70] Peter Van Roy and Alvin Despain, “HIgh-performance Logic Programming with the Aquarius Prolog Compiler,” *Computer*, 25(1):54-68, January 1992

[71] D. E. Thomas et al., “The System Architect’s Workbench,” *Proc. of the 25th Design Automation Conference*, 1988

[72] Jun Sato, et al., “An Integrated Design Environment for Application Specific Integrated Processor,” *Proc. of ICCD*, 1991

[73] “SPEC Benchmark Suite Release 1.0,” October 2, 1989

[74] Ching-Long Su and Alvin Despain, “An Instruction Scheduler and Register Allocator for Prolog Parallel Microprocessors,” *Proc. of International Computer Symposium*, 1992

[75] Ching-Long Su, Chi-Ying Tsui and Alvin Despain, “Low Power Architecture Design and Compilation Techniques for High Performance Processors,” *Proceedings of the IEEE COMPCON*, Febuary 1994

[76] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM J. Res. Dev.*, January 1967, pp. 25-33.

[77] Jean-Paul Tremblay and Paul G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill Book Company, 1985

[78] *Programming Manual for the Teledyne TDY-43 Computer*, Teledyne Systems Company, 1988

[79] Peter L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming*, Ph.D. thesis, Technical Report UCB/CSD 90/600, Univ. of California, Berkeley, 1990

[80] D. F. Wong, H. W. Leong and C. L. Liu, *Simulated Annealing for VLSI Design*, Kluwer Academic Publishers, 1988