University of Alberta

TOWARDS AN ARCHITECTURE FOR IP ROUTING

by

Aaron Richard Dittrich ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95738-1
Our file Notre référence
ISBN: 0-612-95738-1

"Everything that can be invented has been invented."

– Charles H. Duell, Commissioner of US Patents, 1899

Acknowledgements

It is with the aid and support of many people that I saw this project through to completion. First and foremost, I must express my sincere gratitude to my supervisor, Dr. Mike MacGregor. Without his guidance, I would never have made it past the first month. His office door was always open to me, and even when I felt I had made no progress, he was always there to reassure and encourage me. I am especially grateful for his patience while this project continued longer than expected.

Second, I wish to thank Dr. Soner Onder and Peng Zhou at Michigan Technological University. They provided a great deal of help and insight into the FAST system in the early days of my work. Without their help, I would have had much more difficulty in learning the system.

Last, but certainly not least, I owe a great deal of thanks to my parents, Hermann and Erika Dittrich. Without their love and encouragement, I never would have come this far in the first place. They have always shown interest in my work, and have supported me regardless of the paths I have chosen to follow.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the complexity of software increases, so does the hardware on which the software is run. Hardware designers are faced with a constant challenge of providing more features in their products while making them run faster. The demand for processing power seems all but insatiable. On the other hand, hardware designers have anticipated these needs and have begun to provide increasingly sophisticated features before the demand exists. Therefore, when the demand arises, consumers do not need to wait for the next version of the hardware to incorporate them.

Nowhere is this more prevalent than in the microprocessor market. In 1971 Intel introduced their x86 architecture with the Intel 4004, a 4 bit CPU with 46 instructions that ran at 740 kHz. Today's 32 bit Pentium 4 processor, which supports clock speeds greater than 3 GHz, executes multiple instructions per cycle out of order, and boasts a huge instruction set with over 400 operations, 144 of which are meant only for multimedia[1].

Compiler designers initially had trouble incorporating these sophisticated instructions into their compilers, and they did not gain much popularity in the beginning [41]. Changes to the instructions and to the hardware that implemented them eventually allowed easier targeting of the instructions to conventional compilers.

General purpose processors benefit from a variety of such features. Complex instruction sets such as these allow the processor to be targeted to a wide range of applications, such as scientific computing and image processing. There exists a demand, however, for processors that are targeted to very specific applications, and whose instruction sets are optimized for these applications. This is especially the case in the embedded systems market, where programs to be run on the processor are permanently fixed, or where there are very tight constraints on performance, power consumption, or other factors. Many features can be eliminated from the architecture when precise knowledge exists about what operations are required of the processor. Processors that are streamlined for specific applications can have advantages over general purpose processors. These advantages may include lower power consumption, smaller chip area, or reduced circuit complexity.

[1]Originally MMX (Matrix Math Extensions), which later evolved to SSE (Streaming SIMD Extensions), later becoming SSE2 and SSE3

This thesis considers this problem in the context of IP routing. A router can be classified as an embedded system, because it runs a special purpose application program with little or no human intervention, and forms a component of a larger system, the most general case being the Internet. Considering the restrictive nature of the application, the hypothesis is that a simplified processor architecture is sufficient.

1.1 Motivation

As with computer processing power, there is an ever increasing hunger for more network bandwidth. Carriers are faced with increasingly congested links, forcing additional links to be added or faster ones to be put in place. The fastest links commercially available at this time are the OC-768 links, that operate at a rate of approximately 40 Gbps. While naively increasing the speed of links seems like a simple solution, processing the packets as they arrive at the routers remains a significant challenge. Consider a link with a bandwidth of 1 Gbps, a rather common value today. The size of packets can be from 64 bytes upward. Any device processing these packets therefore must be able to handle them frequently enough to allow for a 64 byte size. Given the 1 Gbps link speed, the device will potentially see a new packet arriving every 512 nanoseconds. A commonly held belief is that it takes about 100 cycles to process a packet inside a router. In this case, the cycle time of the processor must be 5 nanoseconds, resulting in a frequency of 200 MHz. This factor remaining the same, for a 40 Gbps link, an 8 GHz processor is required[2].

Given the increasing demand for bandwidth, the next logical step up are links with speeds of 160 Gbps (OC-3072). Before such lightning fast speeds can even approach fruition, serious work needs to be done on how to process so much data so quickly. One avenue of approach is to develop highly specialized processors that can perform these functions very rapidly. According to our results, when using common routing software, the value of 100 instructions per packet is very unrealistic. The actual number is up to 50 times greater! As a result, an examination of both hardware and software performance is critical.

1.2 Goals

Boosting performance of the routing application requires an examination of both the hardware and software architecture, and as such, this work identifies factors from both arenas that have an effect. The major goals of this study are:

- **Develop ISA suitable for routing.** This is the major focus of this work. Given the restricted nature of the routing application, a full-featured general-purpose microprocessor seems unnecessary. Because the Instruction Set Architecture is the interface to the hardware, limiting

[2]Because processors with such clock speeds do not yet exist, router architects have come up with other solutions to this problem.

the instruction set to those operations that are absolutely necessary will reduce hardware complexity.

- **Identify any beneficial hardware changes.** Analyzing how the hardware is used by the software has lead to ideas about how it might be streamlined. A reduction in the length of the critical path through the processor could allow an increase in clock frequency.

- **Identify software issues that are limiting performance.** Software behaviour plays a large role in the overall performance of the routing application. Factors such as memory accesses, data dependencies, and control hazards have a limiting effect on throughput.

The first and second items are significant first steps toward developing a microprocessor specifically for routing. The third is more dependent on the algorithms employed and the behaviour of the compiler, and is the subject of future work in this area.

1.3 Methodology

The architecture chosen as the platform for the tests in this study is the MIPS RISC architecture, specifically the MIPS R2000. The R2000 has become the defacto standard in the embedded computing environment [4]. MIPS processors can be found in everything from palmtop computers to video game consoles to Cisco routers and switches. Programmers and compiler writers enjoy the clarity and flexibility of the MIPS instruction set, and recent versions of MIPS processors provide very competitive performance with modest power consumption. Once the base hardware platform is selected, optimization of the instruction set is an iterative process consisting of the following steps:

- **Profiling.** Benchmark code is executed on the simulated architecture and statistics are gathered about instruction frequencies.

- **Analysis.** By applying design metrics to the statistics gathered, decisions can be made about which instructions are most useful and which can be removed.

- **Modification**. Apply changes to the instruction set, and repeat the process.

The architecture is simulated with the Flexible Architecture Simulation Tool (FAST), which provides an Architecture Description Language in which the pipeline architecture and instruction set of a processor are described. FAST is particularly adept at describing the instruction sets of processors, because all assembler syntax and binary formats are defined explicitly. Pipeline architecture is described as an ordering of stages, and instruction execution is modeled explicitly as a flow of instructions through pipelines. From an ADL description, FAST generates an assembler and cycle-accurate simulator for the target architecture. Statistics are automatically generated about instruction usage, pipeline stalls, and more.

Berkeley Unix routing code is used to gauge the effects of architectural changes on routing functions. This is the code used as the benchmark on the simulated MIPS architecture. Real life routing tables from backbone routers are used as the basis for lookups. The results obtained from the Berkeley code are compared to those from the LC Trie algorithm of Nilsson and Karlsson [29].

Once satisfying changes are made to the instruction set of the processor, a more in depth look at the software behaviour is warranted. The FAST system is again used to analyze the behaviour of the software beyond instruction usage to stall cycles, memory bandwidth, and branches.

The scope of our work is an architectural analysis, which must first take place before any more detailed gate-level analysis of the hardware. Once architectural issues have been identified, a lower-level analysis can determine the exact effects of architectural changes on chip area and circuit complexity.

1.4 Overview of results

Our results can be broken down into the three categories described in Section 1.2. Initial execution of the BSD routing code highlights a number of instructions that are not used at all. These include arithmetic, logic, branch, and load/store instructions. Furthermore, floating point operations are not required, eliminating the need for such a unit. While integer division is used on a very rare basis, it appears to be tied a C library function and not the routing code itself. Elimination of the integer multiplier/divider and its respective instructions could have a significant impact on chip area and complexity. The next step is an examination of instructions that are seldom used and suggestions on how to replace them with pseudo instructions, where possible. Of the eight branch instructions available, only two are actually required. The remaining jump and load/store instructions are difficult to substitute. Due to the limitations of the MIPS ISA, changes to the compiler are required, which is beyond the scope of this thesis. In total, it is possible to eliminate 50% of the instructions from the R2000 instruction set.

Although removing instructions and their functional units has advantages, one bottleneck that remains is the adder. While addition is one of the most prevalent operations, our results show that the majority of addition involves incrementing by one. Although the adder cannot be eliminated completely, a special-purpose functional unit for incrementing may be beneficial. If such a resource could operate at a higher speed, processor frequency might be increased by transforming the general-purpose adder into a multi-cycle functional unit.

One of the major avenues for optimization is the large number of pipeline stalls that occur during execution of the code. Over 40% of all machine cycles are stalls when running the BSD routing code. All of these are a result of dependencies on data being loaded from memory. The values for most instructions do not change significantly with changes to the instruction set, although a few exceptions are discussed. More sophisticated compiler scheduling is required to better cope with the latencies that exist in the MIPS pipeline. Structural changes to the pipeline might also prove useful.

The utility of the simplified instruction set was verified by testing it with the LC trie algorithm. Overall, the instruction usage patterns with the LC trie algorithm were consistent with those of the radix tree algorithm.

1.5 Outline

This thesis is organized as follows. Chapter 2 presents an overview of related work on architecture description languages and instruction set design. Chapter 3 discusses the necessary background for the study, including the MIPS architecture and the basis for the routing performance tests. Chapter 4 presents the Flexible Architecture Simulation Tool, the architecture simulation package used in this study. A discussion of the experiments and their results can be found in Chapter 5, and Chapter 6 concludes with the future directions of this work.

Chapter 2

Related Work

A significant amount of research on instruction set design took place in the early to mid 1990s. Much of this focused on generating instruction set extensions for specific classes of algorithms. Most approaches to the problem are quite similar, and involve searching for frequently recurring groups of operations. The exception to this is the IBM study discussed at the end of this chapter that aims toward as simple an instruction set as possible. Of course, in either case, it is useful to have tools that allow early design space exploration of new instruction sets and processor architectures in general. Architecture description languages are designed just for this purpose and often allow automatic generation of a software toolset.

This chapter begins with an overview of architecture description languages available today and categorizes them according to their major goals. It follows with a survey of related work on instruction set design and discusses design metrics used in the process.

2.1 Architecture Description Languages

As the complexity of processor architectures has grown, so has the time required for the design, testing, and verification process. Architects are faced with shorter time-to-market, an ever increasing number of design choices, and costly manufacturing processes. Waiting until a new architecture is put into silicon is not feasible for testing, for the costs of making changes or correcting bugs could be astronomical. It is for this reason that methods are required for testing and verification earlier in the design cycle.

Architecture Description Languages (ADLs) evolved out of the need for design space exploration of new *Systems On Chip* (SOC) architectures. SOCs have become more common recently as advances in semiconductor technology have allowed integration of processors, memory, and I/O interfaces on a single chip. Traditionally, these items were selected as separate standardized components and connected via a system board to form an embedded system [38]. Now it is possible, however, to place these items on a single chip, which allows a much greater degree of freedom when customizing for a specific application. Designers can target SOCs for specific applications subject to constraints such as chip area, power consumption, and performance. Rather than exploring different architectures at a very low level, ADLs allow specification of *architectural templates* that describe the behaviour of SOCs at a system level. Architectural templates describe the components that reside within the SOC, the interaction among the different components, and how the components themselves function. Low-level details of the hardware are left to later design phases once the SOC has been optimized subject to the specified constraints.

The strengths of ADLs lie in their ability to generate software toolkits for hardware/software co-design. This is made possible by the ability of many ADLs to specify the instruction set architecture (ISA) for the SOC. A compiler and simulator can then be generated based on the architectural template and its ISA. Benchmark programs can then be written in a high-level language such as C and compiled into the native machine code of the SOC. The generated simulator can then be used to estimate performance, analyze resource utilization, determine the suitability of the instruction set, and debug software before the hardware design is finalized.

ADLs are classified into four categories depending on their particular purpose and where their major strengths lie. These categories are *synthesis*, *compiler generation*, *simulator generation*, and *validation*. These categories are not mutually exclusive, as many of the ADLs described below have strengths in more than one area.

2.1.1 Synthesis-oriented ADLs

Synthesis-oriented ADLs allow a designer to convert an architectural template into a low-level processor description. The designer can focus on specifying the blocks and functional units at a high level of abstraction, and the time consuming activity of designing control path structures is automated. Two ADLs that fall into this category are **MIMOLA** and **COACH**.

MIMOLA

MIMOLA is a hardware description language (HDL) developed at the University of Dortmund [7]. Part of a larger suite of utilities, it is designed to support rapid development of VLSI systems by being an input language common to a variety of CAD tools developed at the same institution. The distinction between hardware structure and behaviour is explicitly modeled in MIMOLA. Net-lists of hardware components are used to model hardware structure, whereas Pascal-like HLL code is used to describe hardware behaviour. A disadvantage of MIMOLA is its lack of ability to explicitly describe processor pipelines and resource conflicts.

The MIMOLA language itself does not explicitly support simulator generation, nor does it support the definition of an instruction set or assembly language. Other tools, however, are designed to utilize MIMOLA descriptions for these purposes. The MSSQ compiler [27] reads a MIMOLA hardware description and HLL program and produces architecture-specific binary code. It works by mapping the algorithms to components in the datapath and generating the appropriate microcode.

The RECORD compiler [26] takes a MIMOLA description and a program written in its Data Flow Language, translating it into binary code for the target DSP. RECORD also allows compiler generation by extracting an instruction set from a MIMOLA description. The MSSB/U simulators, part of the MIMOLA Design System, allow simulation of MIMOLA descriptions at both an algorithmic and RTL level.

COACH

COACH is a CAD system developed at Kyushu University that supports hardware, interface, and evaluator synthesis [2]. It performs logic and layout synthesis based on an RT-level HDL hardware description. Using the same HDL code, it then extracts an instruction set for the target architecture and generates a compiler. While COACH supports simulator generation, the resulting simulator operates at an instruction level rather than at a cycle level. It assumes all operations from instruction fetch to execution are one atomic operation, and thus does not support pipelining or instruction level parallelism. Furthermore, the model assumes relatively simple RISC architectures and does not allow superscalar or VLIW processors. Cycle-accurate simulation is available via existing HDL simulators, however these are significantly slower than the simulator generated by COACH.

2.1.2 Compiler-oriented ADLs

Whereas hardware synthesis was the primary goal of the first two systems discussed, generating ILP compilers is the main objective of the following ADLs. Generally these ADLs allow much more detailed description of the instruction set and/or addressing modes of the processor.

nML

nML is an ADL developed at TU Berlin whose primary purpose is to describe the instruction set of a processor [12]. nML is the input to a variety of design tools such as simulators and code generators. It operates at an instruction level, hiding low level architectural details from the user. nML describes instructions and their addressing modes as rules of an attribute grammar. Attributes attached to these grammar rules include behaviour, assembly syntax, and object code image. Because nML operates at an instruction level, it does not have the ability to describe multi-cycle functional units or pipelines. Different methods are available to model delays in the datapath.

ISDL

ISDL is an instruction set description language developed at MIT [14]. It is targeted mainly toward the description of VLIW architectures, although it supports a wide variety of other architectures. ISDL allows automatic generation of an assembler. The ISDL compiler is designed to take a C or C++ program and an ISDL description as input, and output the corresponding assembly code targeted toward and optimized for the specific architecture. The assembler generated from the ISDL description is then used to convert the assembly code into binary machine code for the target. ISDL allows explicit definition of the instruction word format and assembly syntax, as well as the storage available to the processor, such as memory, register files, a stack, etc. Each instruction in the ISA description is separated into fields that define which operations can be performed in parallel. This is why ISDL is particularly suited toward VLIW architectures. Each of these fields includes information on cost and latency, as well as on structural, bitfield, or syntactic constraints. The purpose of ISDL is mainly to describe the instruction set and therefore the pipeline architecture is not explicitly described.

Cycle-accurate simulators can also be generated from ISDL descriptions using the GENSIM tool [15]. These simulators are bit-true and provide a variety of features for debugging and verification of both hardware and software.

MDes

MDes is a machine description language for compilation developed at UIUC [13]. It is used in the Trimaran System [39], a compiler infrastructure for research in ILP architectures. Trimaran provides a cycle-level simulator for its HPL-PD processor family only, and therefore the retargetability of MDes is limited to this architecture. Trimaran provides a compiler front end for C, and a compiler back end that is parametrized by the MDes description and that performs instruction scheduling, register allocation, and compiler optimizations. Memory systems are described explicitly in MDes, however they are limited to traditional hierarchies. In MDes, resource conflicts are captured by explicit description of reservation tables.

EXPRESSION

EXPRESSION is an ADL developed at UC Irvine that supports a wide variety of processor architectures, such as DSPs, VLIW architectures, RISCs, and ASIPs [16]. EXPRESSION is capable of describing both structure and behaviour of processors and has a LISP-like syntax. EXPRESSION operates in two modes. In the *Exploration Phase*, a description is constructed from existing processor and memory libraries. The processor libraries contain a variety of DSP, ASIP, and VLIW processors, while the memory libraries contain caches, SRAM, frame buffers, and a variety of other constructs. Structure is specified as a net-list of these components and the data flows between them. Processor pipelines are modeled as an ordering of stages and the timing characteristics of multi-cycle functional units are also specified. The instructions of the processor are defined in terms of their opcodes, operands, and operations that are performed in each functional unit for the instruction. Once the base description is available, the toolkit generator produces an exploration simulator and exploration compiler from the description that allow comparison of base processors and memory hierarchies. The *Refinement Phase* is entered once the designer settles on a processor and memory hierarchy. In this mode a cycle-accurate simulator and ILP compiler are generated that allow finer tuning of the base processor characteristics and memory system. Resource conflicts are not specified explicitly in EXPRESSION. Instead, reservation tables are automatically generated by extracting information from the structural description.

2.1.3 Simulation-oriented ADLs

The strengths of simulation-oriented ADLs lie in their detailed description of pipeline behaviour and for generation of cycle-accurate simulators. While many of the compiler-oriented ADLs have these features, simulation-oriented ADLs typically do not support compiler generation.

LISA

LISA is an ADL developed at RWTH Aachen for simulation of DSP architectures [33]. It is capable of modeling existing SIMD, VLIW, and superscalar processors, and supports bit-true cycle-accurate simulator generation. Employing a straightforward C-like syntax, LISA models both the structure and behaviour of processors, and supports description of the instruction set. Structure is described with resource declarations that include constructs for registers, memories, and pipelines. These constructs characterize the state of the system at a given moment. As in EXPRESSION, pipelines are modeled as an ordering of stages. The instruction set is described in terms of its assembler and binary syntax, followed by the semantics and behaviour of the instructions. The individual operations performed by each instruction can be described at an instruction, cycle, or phase level. Individual operations are assigned to the pipeline stage in which they should be executed. LISA supports multiple pipelines, and resource conflicts are handled by explicit stalls and flushes of the pipeline. LISA does not support compiler generation, and therefore there is no other explicit method of specifying resource conflicts.

RADL

RADL is an ADL developed at Rockwell Semiconductor Systems that provides pipeline modeling in even greater detail than LISA [35]. Instruction behaviour in the pipelines is partitioned into the phases in which the operations occur. Furthermore, the number of phases per cycle is explicitly specified for each pipeline. It is possible for different pipelines to have a different number of phases per cycle, allowing different pipelines to run at different, but synchronous, speeds. RADL excels at describing inter and intra-pipeline control and data communications. Hazards are handled with strategy tables that specify control signals and their associated actions, such as pipeline stalls, flushes, and instruction kills. RADL also provides support for delay slots, interrupts, and hardware loops. As with LISA, RADL lacks the ability to specify resource conflicts, and there is no means to generate an ILP compiler from an RADL description.

2.1.4 Validation-oriented ADLs

Verification, rather than optimization, is the primary focus of validation-oriented ADLs. Given a processor description, the purpose of these languages is to validate pipeline behaviour such as out-of-order completion and internal data forwarding.

AIDL

AIDL was developed at the University of Tsukuba [28] for describing advanced processors at early stages. Operations in AIDL are synchronized to an implicit clock, where actions occur in intervals between discrete time sequences. As with other ADLs discussed previously, behaviours are described with respect to pipeline stages. AIDL contains two data types. Cause and effect relations between stages are specified with flag variables, while register variables hold data values of one or more bits. AIDL is targeted toward validation of pipeline behaviour, and the AIDL simulator outputs data on the flow of instructions through stages and the values of variables. There is no compiler or synthesis support in AIDL, although it is possible to translate AIDL code into VHDL.

	MIMOLA	COACH	nML	ISDL	MDes	EXPRESSION	LISA	RADL	AIDL
Compilation	√	√	√	√	√	√			
ILP compiler	√		√	√	√	√			
ILP constraints	△					√			
Simulation	△	√	√	√	△	√	√	√	△
Cycle-accurate	△	△		√	△	√	√	√	△
Synthesis	√	√		△					△
ISA			√	√	√	√	√	√	

√ Supported
△ Partial support

Table 2.1: Comparison of ADLs. Source: [38]

2.1.5 Comparison of ADLs

Table 2.1 provides a comparison of the ADLs discussed in this chapter by highlighting some of their main features. Most of the ADLs support compiler generation, with the exception of the last three, which are used strictly for simulation. EXPRESSION is the most sophisticated in terms of compiler generation, while COACH takes a very simplistic view and does not support processor pipelines or instruction-level parallelism. Furthermore, EXPRESSION seems to be the only ADL that provides full support for conflict detection, and automatically generates reservation tables to take care of this. MIMOLA only provides partial support for ILP constraint detection. All of the compiler and simulation-oriented ADLs provide full support for specifying instruction sets.

All of the ADLs provide some level of support for simulation. MIMOLA does not explicitly support simulator generation, but descriptions can be simulated using HDL simulators. The simulation ability of MDes is limited to a particular processor family, while AIDL descriptions need a separate simulator and cannot be used to automatically generate a software toolkit. ISDL, EXPRESSION, LISA, and RADL provide cycle-accurate simulators. The others provide only limited retargetability, or else model behavior at an instruction level only.

MIMOLA and COACH are the only two ADLs that provide effective support for synthesis because they are originally HDLs. Both ISDL and AIDL can be translated into VHDL code, giving them partial support for synthesis.

2.2 Instruction Set Design

Since the advent of digital computers, hardware designers have been constrained by a number of factors. In the early days, factors such as hardware complexity, cost, and reliability were important in computer design. Instruction selection has always played a significant role in both hardware complexity and cost. General purpose computers, which must support a wide variety of functions, require an instruction set flexible enough to meet the needs of a broad range of programmers and their applications. Computers targeted to more specific applications may be able to contend with a more limited set of instructions, reducing the complexity and cost of the hardware. Selecting an efficient and appropriate instruction set depends on [20]:

- The character of the computations
- The frequency of operations
- The flexibility of implementing operations in software using hardware instructions
- The hardware required to implement the instructions
- The performance of the instructions

The character of computations is one of the main factors. Many of today's commercial processors implement complex instructions that modern compilers are only beginning to utilize. Even then, the vast majority of programs are comprised of mostly simple instructions, which is why frequency is also important. If a performance gain is achieved for complex instructions at the expense of the simple ones, it may be more efficient to implement the complex operations in software. This needs to be balanced, however, with the flexibility of implementing these operations in software. It is clear that none of these factors are mutually exclusive. Each has some impact on the others, and it is the task of the designer to choose an optimal instruction set based on these and other factors.

These factors are of course fairly abstract, and designing a real instruction set requires consideration of a number of more specific items as well. The word size and address size are of critical importance. These factors directly influence the resulting code size of programs, as well as the size of the addressable memory space. The word size is affected by the number of registers and the number of operands referenced by instructions. A larger number of registers requires more bits for register IDs, and more operands increases the number of bits in the instruction word. Furthermore, the nature in which these operands are accessed plays a role. If operations can be performed directly on data in memory, the word size may need to be longer to accommodate multiple memory addresses. Because register IDs are typically much shorter, requiring operands to be first placed in registers can shorten the word size. The number and types of addressing modes are also important, and determine the types and lengths of fields in the instruction word.

2.2.1 Instruction set design taxonomy

Holmer [20] discusses a taxonomy of instruction set design whereby instruction sets are generated through an iterative or a constructive process. An iterative process relies on transformations to an initial instruction set to achieve the desired characteristics, whereas a constructive process starts from scratch and builds an instruction set piece by piece. Both these processes can be either instruction-based or feature-based. In an instruction-based process, transformations affect a limited subset of instructions, whereas in a feature-based process, the effects are more widespread across the instruction set.

Iterative techniques

In the iterative, instruction-based techniques, an existing instruction set is normally chosen as the initial base on which to conduct transformations. This is useful when an existing instruction set is to be optimized for a specific application, which is the case in this thesis. Alternatively, instructions from multiple instruction sets can be chosen if one wishes to capture strengths of different architectures. Once the initial instruction set is chosen, transformations are applied to optimize the design metric(s). These transformations can have the effect of deleting, adding, or modifying instructions.

Specialization and *Generalization* are inverses of one another and target an instruction toward a more specific or more general purpose, respectively. In both cases, a single instruction from the original instruction set results in a single new instruction in addition to the existing one. Examples of specialization include reduction in operand size and making an operand value implicit [37, 8]. For example, if incrementing values by 1 is a common occurrence, it may be advantageous to have an instruction specifically for this purpose. Instructions of the form `add Ri,Ri,#1` could be replaced with an instruction like `add1 Ri`, executed by a functional unit designed specifically for this purpose, rather than by a general purpose adder. Holmer also cites a number of studies where specialization can increase performance when data types can be determined at compile time.

The third transformation for iterative techniques is *Decomposition*. This involves replacing one instruction with two or more instructions that, when combined, perform the same operation. This is particularly useful if the resulting instructions are used frequently, but the original is seldom used. In contrast to specialization, decomposition eliminates the candidate instruction and replaces it with simpler instructions. These instructions may be new, or they may already exist in the instruction set. If they already exist, the net result is the deletion of one instruction. For example, the instruction `move Ri,Rj`, where `Ri` and `Rj` contain memory addresses, can easily be implemented with existing `load` and `store` instructions. *Composition* is the inverse of decomposition and combines two or more existing instructions into a single, more complex instruction.

Compaction takes two or more instructions and produces two or more instructions. The idea is to break individual instructions down into their micro-operations, relax the boundaries between these instructions, and rearrange the micro-operations to form new instructions. If the resulting instructions are useful in a broad context and result in some performance improvement, they can be added to the final instruction set. Determining the right mix of micro-operations to achieve this is difficult, however, and therefore the technique has not been widely employed.

The final two methods are less systematic. The *Random* method of instruction selection involves selecting instructions arbitrarily from the universe of all possible instructions, while the *Experience Based* method relies on human knowledge of the application, benchmark execution, and code generation.

Haney and Bose [17, 10] have done work on instruction set design that can be classified as iterative feature-based techniques. These involve construction of an instruction set by manipulating features or characteristics that affect multiple instructions simultaneously, rather than manipulating individual instructions. Haney's system is based on the generalized instruction set, where instructions consist of an operation and one or more features, such as addressing modes or operand fields. Each operation and feature has a cost that is supplied by the user. Different combinations of features and operations result in instructions with different costs. A search program is used to produce an instruction set of maximum value subject to a total cost constraint. Haney's approach suffers from the fact that the costs assigned by the user may not be directly associated with actual hardware and performance costs.

Bose takes a different approach whereby an instruction set is generated based on an attribute grammar of a high level language. The instruction set is generated by applying a set of transformations to the HLL grammar to produce the grammar of the instruction set. Examples of transformations include conversion of the program to tokens, conversion from infix to postfix notation, and adding type information to data values to eliminate a run-time symbol table.

Constructive techniques

With the exception of Holmer's work, very little has been done on generating instruction sets using constructive iteration-based techniques. Holmer's method involves collecting an execution trace from a benchmark program and analyzing the register transfers and the frequency of different operations. The benchmark code is then broken into segments whose starting points are selected at random and whose ending points are chosen so that the segments require roughly the same number of execution cycles. Exhaustive search is then used to determine the optimal sequence of micro-operations to perform the functions of each segment. Instructions are then formed for each of these code segments subject to several constraints. While the instructions generated for each code segment are not candidate instruction sets in themselves, the combination of them is considered to be the final instruction set.

The constructive feature-based techniques are the least studied. It seems that construction of an instruction set purely from features such as operations and addressing modes may not result in an optimal final instruction set, but is more useful for generating an initial instruction set for optimization by another method.

2.2.2 Instruction set design metrics

Holmer discusses a variety of design metrics that have been used in previous studies. Because it is rarely feasible to implement an instruction set in hardware to gauge its performance, these metrics attempt to estimate performance based on a number of different factors. A designer is free to choose one or more of these metrics that they feel are most appropriate for their application. These metrics are discussed below, and their ease of estimation is discussed in terms of the different classes of ADLs.

Execution time/cycle time

Certainly this is one of the most important metrics in any form of hardware design. Estimating execution time directly is often difficult, however. Often it is not clear what effect a hardware change will have on cycle time. With the advent of synthesis-oriented ADLs, this metric may be easier to estimate if a gate-level hardware description can be generated and simulated.

Cycle count

Cycle count is a useful metric if changes to the instruction set do not affect cycle time. If instruction set changes lower cycle count but increase cycle time, however, this metric can be misleading. Cycle count is one of the major metrics used in this thesis, and is easy to determine with many of the ADLs presented earlier that offer cycle-level simulators.

Static code size

Static code size is of more importance in embedded systems where memory is limited than in normal workstations and servers. This metric is directly influenced by factors such as word size, as well as the overall complexity of the instructions available. A more complex instruction set will combine many simple operations into a single instruction, lowering the overall code size, at the expense of more complex hardware. Simple RISC instruction sets, however, may have larger code size with simpler hardware. The compiler-oriented ADLs allow calculation of static code size because benchmark programs can be assembled into binary code.

Dynamic code size

While static code size measures memory storage requirements, dynamic code size measures the number of instruction words fetched from memory. This metric may be important in devices with high memory access latencies or small instruction caches. By reducing dynamic code size, cache hit rates can be increased. This metric is easy to determine with any of the ADLs that offer instruction or cycle-level simulators.

Data memory references

This metric can be used as a measure of execution time, however it has little to do with the instruction set. Far more influential factors are the number of registers in the architecture, and the allocation strategy of the compiler. ADLs that provide detailed information on dynamic code size will provide information about memory references. The sum of this metric and the previous is the total number of words transferred between memory and the processor, and has been used by Bose as a measure of execution time.

Number of words transferred among internal registers

This metric has less to do with the instruction set than with the internal arrangement and connections between the functional units and the registers. Compiler and simulation-oriented ADLs that provide detailed statistics about instruction usage can be used to calculate this metric.

Compile time

While not related to the performance of the hardware, compile time may be an important metric in some instances, such as just-in-time (JIT) compilation. The choice of instruction set may not significantly affect the compile time, however, because it is mostly related to code generation, which is only part of the entire process. Although the compiler-oriented ADLs could be used to estimate this metric, there has been no specific discussion of it.

Hardware design and manufacturing costs

This metric is difficult to quantify. It is safe to say that more complex architectures will require a great deal of verification effort on top of the initial cost of designing and the logic. Furthermore, fabrication techniques greatly influence manufacturing costs, and chip complexity is a function of more than the instruction set. These metrics are also vendor specific, and cannot be estimated with any ADLs.

Chip area

This metric is related in part to the previous two. The area can be affected by the manufacturing technology, and is related to the complexity of the instruction set as well as the number of functional units and the overall complexity of the pipeline logic. Synthesis-oriented ADLs could be used to provide estimates of this metric, given that they generate processor descriptions at a gate-level.

Power consumption

Power consumption is an important metric in mobile devices and embedded systems. An instruction set that minimizes memory size or memory accesses may be of benefit, because the memory system is in large responsible for a bulk of the power consumption. While none of the ADLs discussed in the previous section provide estimates of power consumption, the next version of FAST, presented in Chapter 4, is slated to support this.

2.2.3 The Instruction Set Implementation Method Selection Problem

Instruction set implementation method selection type 1 (ISMP-1) is the problem of deciding an instruction set for an ASIP under the constraints of chip area and power consumption [22]. Imai *et al.* have developed the PEAS system, one component of which is the Architecture Information Generator (AIG). The goal of this component is to decide on an optimum instruction set given the results of profiled application programs and their associated data.

The authors define the term *functionality* to be the union of all operators and functions in the C language. They then divide the functionality into three categories:

- **PRTL**: Primitive RTL refers to functionality that can be implemented with very simple hardware components, such as an ALU and one bit shifter. Examples include integer add and subtract, logical instructions, and logical shift instructions.

- **BRTL**: Basic RTL refers to functionality that can be implemented with combinations of hardware modules, microcode, or in software as combinations of other PRTL and/or BRTL functionality. Examples include integer multiplication and division and rotate instructions.

- **XRTL**: Extended RTL includes all library and user-defined functions, and can be implemented with complex hardware modules such as coprocessors, microcode, or software subroutines.

All primitive functionality is automatically implemented using hardware modules. The goal of the system is to decide which of the remaining functionality to implement and with what method. A balance must be struck between implementing basic and extended functionality using hardware modules and microcode or software. Too many hardware modules lead to increased power consumption and chip area, whereas using too few has a negative impact on performance.

Let n be the total number of functionalities, x_i an implementation method for functionality i, f_i the frequency functionality i is executed, based on the profile data, t_i the execution time of functionality i, a_i and p_i the area and power consumption of functionality i, respectively. Solving the instruction set implementation method selection problem then involves finding a solution vector $X = (x_1, x_2, ..., x_n)$ that minimizes the objective function $T(X) = \sum_{i=1}^{n} f_i \times t_i(x_i)$, subject to the constraints $\sum_{i=1}^{n} a_i(x_i) \leq A\_max$ and $\sum_{i=1}^{n} p_i(x_i) \leq P\_max$.

The ISMP solver takes as input the values of f_i for all i, the parameters A_max and P_max, and a database that includes area and execution time of all functionalities, and outputs a list of optimum implementation methods for the required basic and extended functionalities. The algorithm performs a depth-first search on a search tree where leaf nodes correspond to a choice of implementation methods for the functionalities. Each node maintains information about currently selected functionalities and the partial sum of the objective function. Functionalities are sorted in descending order according to a choice of heuristic functions, and the algorithm examines them in that order. One of the key components of the algorithm is the use of the branch-and-bound technique which uses a tight lower bound calculated at every node to eliminate non-optimal solutions.

Alomary *et al.* have extended this work to consider functional module sharing [3], which occurs when two or more operations can share a common functional unit. By taking this into account, the overall cost in terms of area and power consumption can be reduced over implementing each operation independently. Solving the problem with this additional consideration is the basis of ISMP-2, and amounts to checking whether a choice of hardware module for implementing a particular functionality is being considered the first time. If the hardware module has already been chosen to implement another functionality, its cost is not counted.

The authors claim both algorithms find optimal instruction sets for a variety of problems in a very short time. The branch-and-bound technique is able to constrain the search to less than 2% of the total number of nodes in the tree. The algorithms for ISMP-2 was tested by the authors by finding an instruction set to solve for $\lfloor \sqrt[3]{x} \rfloor$. Functional module sharing was then tested by finding the instruction set for a floating point arithmetic emulator program. Their results show that ISMP-2 has an edge over ISMP-1 due to resource sharing.

2.2.4 Generating instruction set extensions

Huang and Despain propose a method of instruction set synthesis for modern pipelined processors [21]. In their approach, a target architecture is described with a list of micro-operations (MOPs) and their associated parameters, such as resource requirements and timing. Each MOP is assigned a cost in terms of its instruction format and hardware requirements. The algorithm receives benchmarks that are represented by weighted basic blocks in the form of MOP graphs. Constraints on the instruction set, resource constraints, and the objective function are also specified, and the output is an optimal ISA for the target architecture and application.

The semantics of instructions are represented with binary tuples, where the first element is a list of MOPs and the second element a list of fields to be encoded in the opcode of the instruction. Operand encoding is one of the main methods used to combine MOPs into single instructions. Combining MOPs also allows register ports to be unified where possible. The instruction set is generated by solving a modified scheduling problem whereby MOPs of the benchmarks are scheduled into time steps. Each time step corresponds to a single instruction. The algorithm starts with an initial state and then invokes a simulated annealing scheme to modify this state until the objective function is optimized. The annealing algorithm provides a variety of operators to reorganize, combine, and split MOPs both within and between time steps.

The authors tested their algorithm with four different benchmarks ranging from list manipulation to database queries. Three out of four cases, the value of the objective function was better for the automatically generated instruction set than for a manually designed instruction set with the same constraints. In all cases, however, the overall number of cycles required to execute the benchmarks was lower than with the manually designed ISA at the expense of larger instruction sets.

Choi *et al.* have looked at generating application specific instructions for DSP applications [11]. In contrast to the work of Huang and Despain, this method has the ability to generate multi-cycle instructions. Similar to Imai *et al.*, the authors classify instructions into three categories, the P (primitive), C (composite) and S (special) classes. Input programs are again transformed into MOP lists and constraints supplied. The first step consists of matching MOPs to the P-class instructions, which are always included in the instruction set. If the constraints are met with only P-class instructions, then nothing more is done. Otherwise, C-class instructions are generated to try and meet the constraints. If the C-class instructions are still not sufficient, S-class instructions are generated.

The authors concentrate mainly on the methods used to generate single-cycle and multi-cycle C-class instructions. Their goal is to generate a minimal set of C-class instructions that are used frequently in the application and which maximize speed. The problem is formulated as the subset-sum problem, which is the problem of finding a subset of $S = \{x_1, x_2, ..., x_n\}$, where x_i is a positive integer, whose sum is as large as possible but not larger than t. This is adapted to instruction generation by searching for a subset of $S = \{g_1, g_2, ..., g_n\}$, where g_i is the gain achieved by combining a group of dependent MOPs into a single instruction C_i. The sum should be no less than T_d, which is the difference between the execution time with only P-class instructions and the user-given constraint. Although the subset-sum problem has exponential time complexity, the authors employ a polynomial time approximation algorithm.

Generation of multi-cycle instructions involves relaxation of one constraint on MOP selection. In the case of single-cycle instructions, a candidate group of MOPs had to be executable in a single cycle, but for multi-cycle instructions, this restriction is lifted. The maximum number of cycles allowed for multi-cycle operations can be varied in order to control instruction set size and to minimize the objective function.

Atasu *et al.* [5] tackled the problem of instruction set extension by directly examining HLL source for recurring groups of operations. They model data flow within a basic block i as a graph $G_i^+(V \cup V^+, E \cup E^+)$, where V is the set of primitive operations in the basic block, and E the data dependencies. V^+ is the set of input and output variables, and the edges E^+ connect the input and output variables with the other nodes. Their technique examines basic blocks and attempts to find optimal cuts S, where $S \subseteq G_i^+$ for a basic block i. The value of a cut S is judged by an objective function $M(S)$, which is usually some estimate of the performance gain achieved by combining the primitive operations of S into one instruction. A cut S is deemed valid only if it satisfies three constraints, the first being that the number of input variables be less than or equal to a user-defined value of N_{in}, the second that the number of output variables be less than or equal to N_{out}, and the third that S is convex. Convexity holds if there exists no path from $u \in S$ to $v \in S$ that has intermediate node $w \notin S$. Their algorithm attempts to maximize $\sum_j M(S_j)$ under these three constraints for each basic block by finding up to N_{instr} cuts S_j that provide the best performance gain.

Because there are $2^{|V|}$ possible cuts in each basic block, examining each possibility is not efficient. The authors describe an algorithm that identifies each valid cut while eliminating those regions that are invalid early in the search. Nodes in the graph are subjected to a topological sort that generates an implicit search tree that is explored recursively. Each level i in the tree represents node i in the topological ordering. The right child (1-branch) of a node at level i represents the addition of node $i + 1$, while the left child (0-branch) represents no addition of that node. At each stage of a pre-order traversal, the three constraints are checked. If any of them fail, the subtree rooted at that node can be pruned. The best cut is selected from those nodes that represent valid cuts. The authors describe an iterative technique for finding up to M optimal cuts per basic block by extending the search tree to support $(M + 1)$-way branching.

Praet *et al.* [34] have done similar work on instruction selection for ASIPs by combining micro-operations with a technique they call *bundling*. A bundle is defined to be a "maximal sequence of micro-operations in which each micro-operation is directly coupled to its neighbours." Direct coupling occurs when primitive activities pass data to one another through resources such as wires or latches. Instruction selection by bundling consists of grouping data flow operations of an application into bundles, and then combining control flow operations and bundles into instructions. Their methods follow a similar approach to the others, in that statistical information is extracted from data flow graphs to find frequently occurring groups of operations.

Lee, Choi, and Dutt have looked at instruction selection for RISC-based configurable processors [25]. Their approach focuses on ISA specialization with only minor changes to the existing data path architecture, and tries to use the existing resources more efficiently for a specific application. In addition to taking an application program as input, their approach also takes as input the data path architecture, producing a basic instruction set and a set of C-instructions. These C-instructions are application specific and are synthesized from an analysis of the application and data path architecture, whereas the basic instruction set is common across all implementations. Each basic instruction consists of a single micro-operation, and the C-instructions amount to combinations of basic instructions that provide performance improvements when combined into one instruction, while using existing hardware resources.

A particular strength of this framework is its ability to address the problem of multiple instruction formats. The synthesis process determines opcode and operand field widths, and allows multiple formats to fully realize the potential of limited bit-width instructions. In addition, the concept of operand class is integral to the system. Operand classes are defined by their field widths, compatible operands, and encoding. For example, immediate constant 4, 4-bit immediates, or 6-bit immediates could all be different operand classes, providing different degrees of generalization for synthesized C-instructions.

The synthesized instruction set consists of the basic instruction set along with the generated C-instructions. The first step is to generate all possible C-instructions that have a performance benefit in terms of cycle count. The best candidates are then chosen by weighing their benefits and costs in an optimization problem. Generation of C-instructions can be further divided into two stages: rescheduling and generalization. Rescheduling involves building all possible C-instructions from basic instruction sequences up to a specified length, provided the C-instruction provides some benefit. Generalization involves application of the different operand classes to the C-instructions to create multiple versions of the instruction. From all these C-instructions, the final set is chosen.

The authors present both integer linear programming and heuristic algorithms for choosing the final set of C-instructions. The selection problem amounts to maximizing the cycle count reduction subject to the code space constraint. The code space constraint can be expressed as $\sum x_i \times 2^{W_i} \leq CS^C$, where W_i is the bit-width needed for the operands of C_i, CS^C is the total code space available for C-instructions, and x_i is 1 if C_i is selected, 0 otherwise. The total codespace available to C-instructions is defined to be $2^{IBW} - \sum 2^{B_i}$, where IBW is the instruction width (typically 16 or 32), and B_i the number of operand bits required for each basic instruction.

The authors conclude with a performance evaluation of their framework when applied to the MIPS architecture. They observe performance gains of 20 - 40% for multimedia applications when using the synthesized instruction set instead of the native instruction set. For other benchmarks, the performance gains are smaller but consistently positive.

While most work on instruction set extensions has been targeted toward increasing performance, Biswas and Dutt have considered it in the context of reducing static code size for VLIW DSPs [9]. Specifically, their approach considers heterogeneous-connectivity-based DSP architectures. Simply put, this refers to limited connectivity between functional units and register files. The example architecture they use is the TMS320C6xx by Texas Instruments, which issues up to eight instructions in parallel, and has two clusters of four functional units, each cluster having its own register file. The goal of their work is to reduce code size without sacrificing performance by clustering instructions that can be dispatched in parallel. Only minor changes are required to the decoding hardware because a complex instruction $B1; B2$ is expanded into its constituents $B1$ and $B2$ during decoding, and each separate instruction is sent to its respective functional unit. In order for instructions to be clustered, there cannot be any contention for resources between them, although they may be dependent on one another for data. Greedy and heuristic algorithms are presented that find candidate clusters subject to connectivity, latency, and dependency constraints. The end result is a set of four-operand two-operator complex instructions that provide a roughly 25% reduction in code size across a range of DSP benchmarks.

2.2.5 Reconfigurable hardware

Athanas and Silverman have devised a system that automatically identifies and synthesizes instruction set extensions during program compilation [6]. The difference between their approach and that of others is the use of Field Programmable Gate Arrays (FPGAs) that allow automatic reconfiguration of the processor architecture. Extended operations are automatically synthesized and implemented in the reconfigurable hardware. The core of their PRISM system is a *configuration compiler* that produces both a hardware image and software image from an application program written in C. The hardware image is used to program the reconfigurable platform, whereas the software image is the machine language version of the application program that utilizes instructions from the processor core and the reconfigurable hardware. The authors provide very little detail on how the extended operations are extracted from the application program at compile time. They note that the time required to execute operations on the reconfigurable hardware plus the transit time to and from the processor core must be less than computing the same operation in software, otherwise no improvement will be observed. With an initial implementation of the system, they observed speedup factors of between five and 54 times for various functions when implemented in hardware.

2.2.6 ISAs for routing in multiprocessors

In 1995, a study was performed at IBM to determine a flexible router architecture that would support a variety of oblivious routing algorithms for interconnection networks [32]. The architecture was required to support fixed and reconfigurable topologies, and the topologies considered were trees, meshes, cubes, and multistage interconnection networks. The goal of the study was to define an instruction set that was as small as possible that could implement the routing algorithms considered, as well as provide handshaking functionality.

The authors begin by describing the structure of their router architecture, where the general components are n input controllers, n output controllers, and the switching mechanism. The input controllers are responsible for executing the routing algorithm, and the output controllers for sending the switched packet to the next input controller. Input controllers consist of a port controller, a routing algorithm handler, and a packet flow controller, and the ability of the routing algorithm handler to support multiple routing algorithms is where most of the router's flexibility lies.

To support the functionality of the routing algorithm handler, the authors identify a general instruction set and a control instruction set. The general instruction set consists of twelve ALU, shift, and control instructions. Instructions in the general ISA are 32 bits in length and all operate on data in registers or on immediate values. The general instruction set employs only absolute addressing as its addressing mode, because the only instruction that performs a memory reference is the branch instruction. The control instruction set is not used for the routing algorithms directly, but rather for initialization, operations with the local processor, or advanced algorithms. Table 2.2 summarizes the two instruction sets.

	Description
General instruction set	
add R1, R2, R3	Addition
sub R1, R2, R3	Subtraction
cmp R1, R2	Comparison
and R1, R2, R3	Logical AND
xor R1, R2, R3	Exclusive OR
plo R1, R2	Position of leading 1 bit
shr R1, R2, R3	Shift right
shl R1, R2, R3	Shift left
mov R1, R2	Register move
bc Mask, Address	Branch on condition
out Channel	End of program
msg R1, R2, R3	Message to local processor
Control instruction set	
lpg R1, R2, R3	Load program
lsr R1, R2	Load status register
lr R1, R2	Load general register
ecp Address	End of control program

Table 2.2: Oblivious routing instruction set

The study considers 40 different interconnection network topologies. The number of instructions needed to implement the routing algorithms for these topologies range from five to twelve. Consider the hypercube topology shown in Figure 2.1. In an n-dimensional hypercube, each node is directly connected to n other nodes, and there are 2^n nodes in total. Each node is given a unique address that corresponds to a string of n bits. Furthermore, the address of each neighbor of a node differs from the node's address by 1 bit. The bit in which it differs corresponds to a particular direction the neighbor lies in, in this case (x, y, z).

The routing task for this topology involves comparing the address of the current node to that of the routing tag, and making a decision along which path to forward the packet. The program can be summarized as:

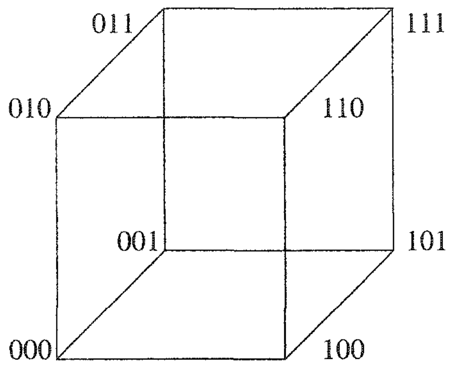

Figure 2.1: 3 dimensional hypercube network

```
            R1: Routing tag;
            C1 = 0;
            C2 = 1;

            cmp R1, C1;
            bc  1000, processor;
            plo R1, R2;
            shl C2, R2, R3;
            xor R3, R1, R1;
            out R2;
processor:  out 4;
```

The value in register R1 is the routing tag, C1 and C2 are the constants 0 and 1, respectively. The first action is to check whether the tag equals zero. If so, the program branches and sends the packet to the local processor (direction 4). Otherwise, the `plo` instruction finds the position of the leading 1 bit, which is stored in R2. The constant 1 is then shifted left by that many positions, and then an exclusive OR is performed with that value and the routing tag. This has the effect of turning off that bit. Finally, the packet is routed in the corresponding direction.

The study also presents a number of examples with more complex topologies. In all cases, however, the instruction set required is comparable in size to that for the hypercube. The study does not discuss the methodology used to synthesize the instruction sets. It likely involves a significant amount of human experience.

2.3 Concluding remarks

A variety of ADLs have been discussed, each with their particular strengths and weaknesses. The synthesis-oriented ADLs are more closely related to hardware description languages, given their very low-level description of processor architectures. Their strengths lie in their ability to automatically generate processor descriptions, saving much of the time required for logic design. The compiler-oriented ADLs are particularly strong at describing the instruction set architecture of processors. These descriptions are then used to automatically generate ILP compilers for the target architecture. Most of the compiler-oriented ADLs also provide cycle-level simulators, making them excellent candidates for precise and comprehensive architectural design and testing. The simulation-oriented ADLs provide even more sophisticated pipeline modeling, but lack the ability to generate compilers. The only ADL in the validation-oriented category seems to be targeted toward validation of internal data forwarding and out-of-order completion, and has no compiler or synthesis support. An ideal ADL would combine the strengths from each of these four categories. Of the ADLs surveyed, EXPRESSION seems to come closest to doing this.

Iterative instruction-based techniques are the most widely studied methods of instruction set design. These techniques involve the application of various transformations to instructions in order to modify the ISA. Modifications are done iteratively until a final instruction set is reached. The iterative feature-based techniques, on the other hand, manipulate features of the entire instruction set to craft it into a usable form. The constructive techniques have not been studied as much. Holmer did work on iterative construction of an instruction set by examining benchmark source code. Construction of an instruction set from features does not seem ideal for generating a final instruction set.

Most of the related work on instruction set design falls into the category of iterative instruction-based techniques. Most involves generating application-specific instruction set extensions for existing architectures. The techniques are are all quite similar, in that benchmark programs are examined to find frequently recurring groups of micro-operations that qualify as candidate instructions. The only study that significantly differs is that done by IBM, whereby the simplest possible instruction set was derived, which has the most relevance to this thesis.

Chapter 3

Background

This chapter introduces the RISC design methodology and describes the MIPS R2000 RISC architecture, which is the basis for the experiments conducted in this study. The routing table mechanism used in the 4.3 Reno release of Berkeley Unix is also described.

3.1 Reduced Instruction Set Computers

Reduced Instruction Set Computers evolved in the early 1980s as a response to the increasingly complex processor architectures appearing at the time. As advances were made in VLSI design, processors began supporting increasingly complex instructions. RISC processors, on the other hand, are based on the idea that programs consist mainly of simple instructions, with very few complex instructions. A RISC processor is designed in such a way that the simple instructions execute very quickly, at the expense of a few complex instructions being slow. The goal in a RISC architecture is for the processor to retire one instruction per cycle. In contrast, Complex Instruction Set Computer (CISC) processors have a large number of complex instructions. This has a performance impact on the simple instructions as well, often requiring several cycles per instruction.

RISC architectures have a number of benefits for processor architects [24]. Implementation of RISC architectures is more straightforward than for CISC processors . The simpler and streamlined design allows for faster debugging and verification of the architecture. This shortens the design cycle, allowing processors to go to market sooner, and aids in the exploration of different design options. A simpler architecture also translates into a smaller chip area. Advantages of this include more area for other resources such as register files or caches, the potential for lower power requirement when these processors are used in battery-powered devices, or performance increases from a shorter critical path through the processor.

RISC processors provide a number of benefits for programmers and compiler designers as well. For those programming at the assembly level, the uniform and much simpler instruction set of a RISC processor is far simpler to learn and use. As an example, the Pentium instruction set contains over 400 instructions, whereas the MIPS instruction set (not including coprocessors) contains less than 75.

3.2 The MIPS R2000/R3000

The MIPS R2000/R3000[1] architecture is a fully 32 bit pipelined architecture. The R2000 is a *load-store* architecture, meaning that all operations are performed on data in registers, and the only instructions that can access memory directly are load and store instructions. All R2000 registers are 32 bits wide, and all instructions consist of one 32 bit word. Including the system coprocessor that is part of the R2000, four additional coprocessors are supported. The R2010/R3010[2] floating point unit is commonly found alongside the R2000 processor in systems where floating point operations are required.

3.2.1 Memory and registers

The MIPS memory space is divided into the kernel address space and the user address space. The kernel address space begins at the top of memory and continues until address `0x80000000`. The user address space is composed of three areas. The *text segment* holds the actual program instructions, and is the location from where the program fetches instructions. This address space begins at address `0x00400000` and ends at `0x10000000`. The *stack segment* grows from the top of the user address space (`0x7fffffff`) down toward the *data segment*, which has no fixed boundary.

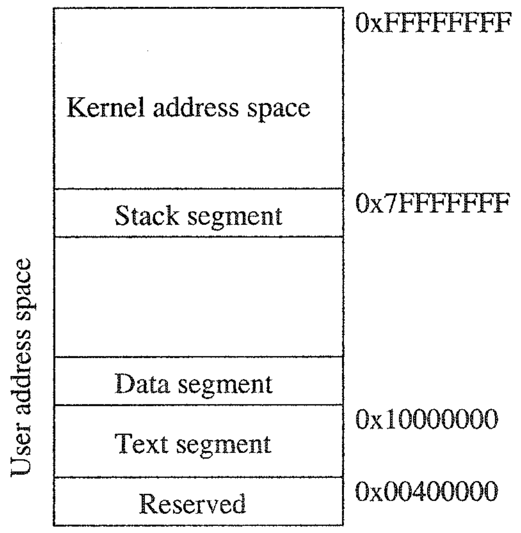

Figure 3.1: MIPS memory layout

The R2000 contains 32 general purpose registers. These registers are summarized in table 3.1. While the hardware does not enforce the usage of registers, each register has an assigned purpose

[1]The R2000 and R3000 architectures are identical in behaviour and structure except for the R3000's increased speed. From now on, R2000 can be taken to mean both.

[2]Likewise for the R2010.

and the programmer must be aware of these in order to ensure that software functions correctly.

Register name	*Number*	*Usage*
`zero`	`$0`	Constant 0
`$at`	`$1`	Reserved for assembler
`$v0 - $v1`	`$2 - $3`	Procedure return values
`$a0 - $a3`	`$4 - $7`	Arguments to procedures
`$t0 - $t7`	`$8 - $15`	Temporary
`$s0 - $s7`	`$16 - $23`	Saved temporary
`$t8 - $t9`	`$24 - $25`	Temporary
`$k0 - $k1`	`$26 - $27`	Reserved for OS kernel
`$gp`	`$28`	Pointer to global area
`$sp`	`$29`	Stack pointer
`$fp`	`$30`	Frame pointer
`$ra`	`$31`	Return address

Table 3.1: MIPS registers

Register 0 retains a value of 0 regardless of what value is written to it. It is used for comparisons to zero. Register 1 is reserved for use by the assembler, such as when an intermediate value must be stored in the expansion of pseudo instructions. Registers 2 and 3 are used to return values from procedures, while registers 4 to 7 are used to pass arguments to procedures. If a procedure requires more than four arguments, the remaining ones are passed on the stack. Registers 8 to 15, 24 and 25 are used for temporary storage of values that do not need to be preserved across procedure calls, whereas registers 16 to 23 are used for temporary storage of values that need to be preserved across calls. Registers 26 and 27 are for use by the operating system. Register 28 holds the address of the middle of a 64K block of memory in the data segment where constants and global variables are stored. The top of the stack is pointed to by the address contained in register 29. Register 30 points to the area of memory containing a procedure call frame. The return address of a procedure call is stored in register 31.

3.2.2 Pipeline Architecture

The R2000 pipeline contains five stages and an instruction can pass through the pipeline in five cycles. While stalls due to data dependencies and memory latency complicate matters, the goal is to retire one instruction per cycle, which requires five instructions in flight through the pipeline at any given time. Each stage is supplemented with a set of pipeline registers where operands and results are stored after the stage operates on the instruction.

The first stage of the R2000 pipeline is the *Instruction Fetch* (**IF**) stage. This stage is mainly responsible for sending the program counter (`PC`) to memory in order to fetch the next instruction word, and then incrementing the program counter. The new instruction is stored in the instruction register (`IR`), while the incremented program counter is stored in the register `NPC`.

The second stage is the *Instruction Decode* (**ID**) stage. This stage does exactly what its name suggests, which is to decode the instruction word, fetch data from the input registers, and determine the destination register. Due to the simplicity of MIPS instruction formats, it is possible to read the input registers at the same time the instruction is being decoded. This is called *fixed-field decoding* since the input fields are fixed bit fields in the instruction words. The data accessed from the input registers are loaded into temporary pipeline registers for use by subsequent stages. In the event that it is needed later, the sign-extended immediate is also calculated during this stage since the immediate portion of an instruction is always located in the lower 16 bits of `IR`.

It is in the **ID** stage that the R2000 determines whether an instruction is a branch instruction, and if so, computes the effective address. The effective address is computed by adding the sign-extended immediate, shifted left two bits, to the address stored in `NPC`. The input registers are then checked to determine whether the branch is taken, and if so, the program counter is modified to reflect the branch target. Fetching then resumes from the target address. Because branches are resolved in the **ID** stage, they incur a penalty of only one clock cycle. This is discussed in more detail in Section 3.2.5.

Based on the type of instruction, the **ID** stage decides where to send the instruction next. If the instruction requires the integer unit, it will be sent to the **EX** stage. The other options are either the system control coprocessor or another coprocessor, such as the R2010 floating point unit.

The third stage in the R2000 pipeline is the *Execution* (**EX**) stage. It is in this stage that the ALU operates on the operands. The operation can be either a memory access, a register-register instruction, or a register-immediate instruction. In all cases, a calculation is done with the operands loaded by **ID**, and the result of the operation is stored in the pipeline register `ALUOutput`.

The fourth stage is the *Memory Access* (**MEM**) stage, and only handles loads and stores. All other instructions pass through this stage unaffected. The effective address computed in **EX** and stored in `ALUOutput` is used to reference the correct memory location. In the case of a store, the value is written to memory at that location. For a load, once the value returns from memory, it is placed in the `LMD` register for later use.

The fifth and final stage is the *Write-Back* (**WB**) stage. This stage simply writes the results of any previous computations or memory accesses into the register file. Instructions that do not write to any registers, such as branches and stores, pass through this stage unaffected.

3.2.3 Instruction Set Architecture

All MIPS instructions consist of a single 32-bit word aligned on a word boundary. There are three MIPS instruction formats for integer instructions. These are displayed in Figure 3.2, and consist of the *Register*, *Jump*, and *Immediate* formats. Table 3.2 summarizes the MIPS instruction set, not including coprocessor instructions and floating point instructions. The instructions are divided into six categories.

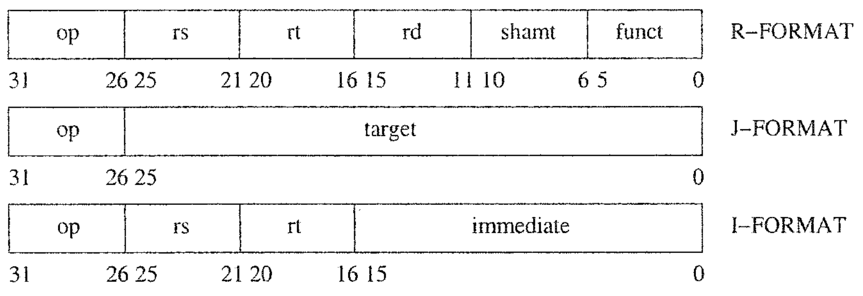

Figure 3.2: MIPS Instruction Formats

Load/Store instructions

The purpose of load and store instructions is to move data between general purpose registers and memory. All load and store instructions are of the Immediate instruction format, and with the exception of the `lwl` and `lwr` instructions, all load instructions have a latency of one cycle.

Computational instructions

Computational instructions include those from the **Shift**, **Arithmetic/Logic**, and **Multiply/Divide** categories. All instructions in this category operate on values in registers, and occur in both Immediate and Register formats.

Jump and Branch instructions

Jump and branch instructions are used to modify the control flow of a program. Jump and branch instructions introduce a delay of one instruction. After the new program counter is computed, the instruction following the branch is executed while the target instruction is being fetched.

Jump and branch instructions occur in all three MIPS instruction formats. The Immediate format is used for branches, *jump-and-link* and *branch-and-link* instructions. The Register format is necessary for the *jump register* instruction, where an address is contained in a register. The Jump format is used for the remaining jump and *jump-and-link* instructions.

3.2.4 MIPS coprocessors

The MIPS R2000 architecture supports up to four coprocessors, designated `CP0` through `CP3` that can interface seamlessly with the R2000 main execution unit. These coprocessors perform ancillary tasks such as exception handling, or operate on other types of data such as floating point numbers.

The System Control Coprocessor (CP0) is located on the R2000 chip. It provides a number of registers and functions for handling exceptions and the virtual memory system. As with most modern processors, the R2000 supports a large virtual memory address space. The Translation Lookaside Buffer on CP0 handles virtual memory mapping. The virtual memory system provides up to 2 GB for users and 2 GB for the kernel. The R2000 enters kernel mode when an exception

OP	*Description*	*OP*	*Description*
	Load/Store Instructions		**Shift Instructions**
`lb`	Load Byte	`sll`	Shift Left Logical
`lbu`	Load Byte Unsigned	`srl`	Shift Right Logical
`lh`	Load Halfword	`sra`	Shift Right Arithmetic
`lhu`	Load Halfword Unsigned	`sllv`	Shift Left Logical Variable
`lw`	Load Word	`srlv`	Shift Right Logical Variable
`lwl`	Load Word Left	`srav`	Shift Right Arithmetic Variable
`lwr`	Load Word Right		**Special Instructions**
`sb`	Store Byte	`syscall`	System Call
`sh`	Store Halfword	`break`	Break
`sw`	Store Word		**Multiply/Divide Instructions**
`swl`	Store Word Left	`mult`	Multiply
`swr`	Store Word Right	`multu`	Multiply Unsigned
	Arithmetic/Logic Instructions	`div`	Divide
`add`	Add	`divu`	Divide Unsigned
`addi`	Add Immediate	`mfhi`	Move From HI
`addiu`	Add Immediate Unsigned	`mflo`	Move From LO
`addu`	Add Unsigned	`mthi`	Move To HI
`and`	AND	`mtlo`	Move To LO
`andi`	AND Immediate		**Jump and Branch Instructions**
`lui`	Load Upper Immediate	`j`	Jump
`nor`	NOR	`jal`	Jump And Link
`or`	OR	`jr`	Jump to Register
`ori`	OR Immediate	`jalr`	Jump And Link Register
`slt`	Set on Less Than	`beq`	Branch on Equal
`slti`	Set on Less Than Immediate	`bne`	Branch on Not Equal
`sltiu`	Set on Less Than Immediate Unsigned	`blez`	Branch on Less than or Equal to Zero
`sltu`	Set on Less Than Unsigned	`bgtz`	Branch on Greater Than Zero
`sub`	Subtract	`bltz`	Branch on Less Than Zero
`subu`	Subtract Unsigned	`bgez`	Branch on Greater than or Equal to Zero
`xor`	Exclusive OR	`bltzal`	Branch on Less Than Zero And Link
`xori`	Exclusive OR Immediate	`bgezal`	Branch on Greater than or Equal to Zero And Link

Table 3.2: MIPS R2000/R3000 Instruction Set

is detected. The *restore from exception* (`rfe`) instruction returns the processor back to user mode. Examples of exceptions include system calls, I/O interrupts, and arithmetic overflows. Memory mapping differs between the two modes, although each is allocated an address space of 2 GB. While in User mode, the R2000 can only access the user address space, whereas when in Kernel mode, references can be made to both the Kernel and User address spaces.

In order for the R2000 to support floating point operations, a floating point accelerator is typically implemented as coprocessor 1 (CP1). This unit provides a separate pipeline and independent register file to that of the main execution unit in the R2000. Execution of floating point instructions in the floating point unit takes place in parallel with execution in the main unit. The floating point accelerator offers its own instruction set that can perform load and store operations, moves, and register to register floating point operations.

Moving data from memory to coprocessors and between coprocessors and the main processor is accomplished with a set of instructions for this purpose.

`lwcX`	Load Word to Coprocessor X
`swcX`	Store Word to Coprocessor X
`mtcX`	Move word To Coprocessor X
`mfcX`	Move word From Coprocessor X

Table 3.3: Coprocessor Load/Store/Move Instruction Set

The load and store instructions move data between coprocessors and memory, while the move instructions move data between coprocessor and the main processor.

3.2.5 Jumps and Branches in the R2000

MIPS implements delayed jumps and branches. This means that the target address is not resolved until later in the pipeline, and therefore a delay slot exists immediately following the jump or branch instruction. The processor always executes the instruction immediately following the jump or branch while the target instruction is being fetched from memory. This is an alternative to stalling the instruction fetch stage while waiting for the target to be resolved. Such a system would require more complex pipeline logic, and would have a negative impact on performance. The alternative is to delegate the task of handling this delay to the software. MIPS assemblers are designed in such a way that this delay slot is filled with a usable instruction, one that should be executed whether the branch is taken or not. If such an instruction is not available, the delay slot is filled with a `nop`. Consider the following fragment of code:

```
          lw     $8, 10($sp);
          beqz   $8, target;
          sw     $8, 20($sp);
          sll    $10, $8, 2;
          .
          .
          .
target:   sll    $10, $8, 4;
```

The machine loads the word 10 bytes above the stack pointer, stores it in `$8`, and then performs a comparison to determine whether the value of that word equals zero. Prior to the branch being resolved, the store word instruction has already been fetched from memory and is inserted into the pipeline. If the branch is resolved as taken, the program counter is then modified to point to the branch target. If the branch is resolved as not taken, the program counter is left unchanged. In both cases, the `sw` instruction proceeds normally through the pipeline.

Figure 3.3 displays the timing diagram in the case that the branch is resolved not taken. Instructions are fetched in sequence and control proceeds normally. Figure 3.4 shows the change in control flow when the branch is resolved as taken in **ID**. By this time, `sw` has entered the pipeline. The program counter is then modified and fetching resumes from target in the next cycle.

lw \$8, 10(\$sp)	IF	ID	EX	MEM	WB			
beqz \$8, target		IF	ID	EX	MEM	WB		
sw \$8, 20(\$sp)			IF	ID	EX	MEM	WB	
sll \$10, \$8, 2				IF	ID	EX	MEM	WB

Figure 3.3: Timing diagram when branch not taken

lw \$8, 10(\$sp)	IF	ID	EX	MEM	WB			
beqz \$8, target		IF	ID	EX	MEM	WB		
sw \$8, 20(\$sp)			IF	ID	EX	MEM	WB	
sll \$10, \$8, 4				IF	ID	EX	MEM	WB

Figure 3.4: Timing diagram when branch taken

3.3 Routing table lookup

Routing table lookup is the predominant activity of a router and is the function for which we wish to find an optimal architecture. The basis for our tests is the routing code from the 4.3 Reno release of Berkeley UNIX. The results from this algorithm are then compared to those from the LC trie algorithm.

3.3.1 Radix tree data structure

The 4.3 Reno release of Berkeley UNIX uses a modified PATRICIA tree to store routing table information and perform lookups [36]. A PATRICIA[3] tree is a variation of a binary radix tree where all nodes with one child are merged with their parent. In a normal binary radix tree, each node along a branch represents a successive position to test in the key. This is an effective strategy when the set of keys is large. When the set is sparse, however, most of the internal nodes have only one descendant, which is not very efficient. To combat this problem, a PATRICIA tree uses selective bits to determine branching.

Consider the Figure 3.5 where the tree contains two keys ababb and abaaa. Rather than having a tree of depth six by branching on each position in the key, it is possible to have a tree of depth two by examining the longest matching prefix and then branching on the value of the next

[3]Practical Algorithm To Retrieve Information Coded in Alphanumeric

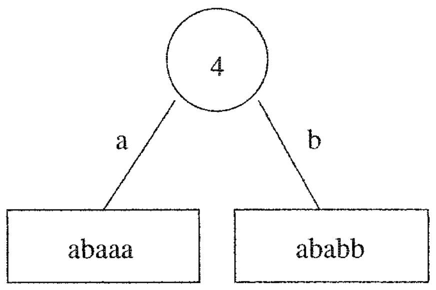

Figure 3.5: Simple PATRICIA tree

character, which in this case happens to be at position four. Suppose now that the value aa needs to be added. This value differs from the others at the second position, so the tree will need to be rearranged. The result is depicted in Figure 3.6.

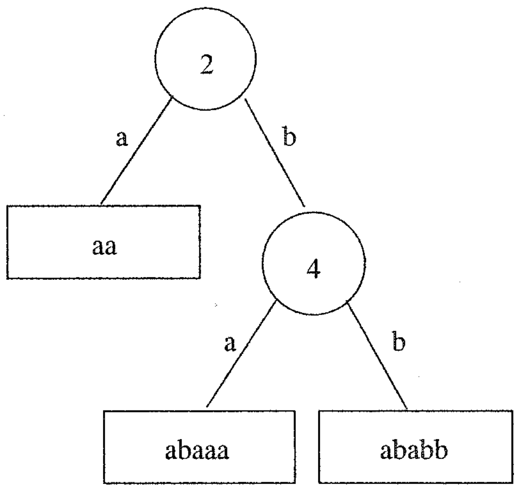

Figure 3.6: Tree after adding aa

The second position is now tested first. If the character is an a the left subtree is chosen, if the character is a b the right subtree is chosen, where the fourth bit is then tested and the process repeated.

The tree for the routing table is constructed in a similar manner. Leaf nodes in the tree correspond to address classes or routes, and internal nodes determine the bit positions to test. A route consists of a prototype address and a mask (the class), as well as other information such as interface number, next hop, etc.

Consider the routing table in Table 3.4. The corresponding tree is pictured in Figure 3.7. If the table is constructed in the order the routes are listed, the first bit to be checked will be bit zero, since this is where all the routes differ from the default. At this point, the default route will be the left branch, and PNet will be the right, since its 0 bit is on. The next bit position to test is determined to be bit nine, since this is where HNet differs from PNet. PNet becomes the left child, while HNet

Prototype	Mask	Route
0	0	Default
0x80050000	0xffff0000	PNet
0x80500000	0xffff0000	HNet
0x80502400	0xffffff00	ZNet
0x80507600	0xffffff00	QNet

Table 3.4: Example routes

becomes the right child of the second internal node. To add ZNet, an internal node to test bit 18 is added. Finally, when QNet is added, the tree is rearranged so bit 17 is tested third, whose left subtree tests bit 18, and whose right child is QNet.

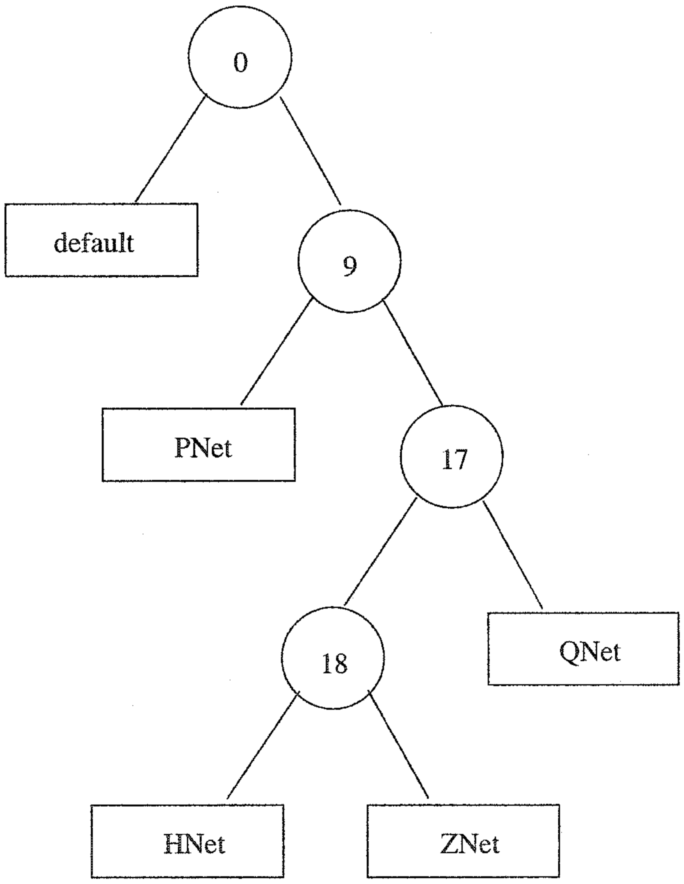

Figure 3.7: PATRICIA tree for routing table 3.4

3.3.2 Radix tree lookup Algorithm

To determine whether a candidate address belongs to a class, the address is compared to the prototype address of that class by computing the exclusive OR of the two addresses. The incoming address belongs to the class if and only if all bits that are set in the result are clear in the mask, because they must then be set in the incoming address. If any set bits in the result correspond to set bits in the mask, they must be clear in the incoming address and therefore the address does not belong to the class. Consider the candidate address 0x80507619. In the case of the classes in Table 3.4, the address belongs to HNet, QNet, and Default, but not PNet or ZNet. The binary results of the XOR and the binary masks are shown in Table 3.5.

Class	XOR Result	Mask
PNet	0000 0000 0101 0101 0111 0110 0001 1001	1111 1111 1111 1111 0000 0000 0000 0000
HNet	0000 0000 0000 0000 0111 0110 0001 1001	1111 1111 1111 1111 0000 0000 0000 0000
ZNet	0000 0000 0000 0000 0101 0010 0001 1001	1111 1111 1111 1111 1111 1111 0000 0000
QNet	0000 0000 0000 0000 0000 0000 0001 1001	1111 1111 1111 1111 1111 1111 0000 0000

Table 3.5: Binary results of XOR operation

The goal of the lookup algorithm is to determine which class an address belongs to. This involves finding the class whose prototype address has the longest matching prefix with that of the candidate address, subject to the mask of the class. To find the appropriate class in the tree, the following algorithm is used:

```
Current node ← Root
while (Current node is not a leaf node) do
  Extract bit position to test
  if (Bit of candidate address is on) then
    Current node ← Right child of Current node
  else
    Current node ← Left child of Current node
  end if
end while
```

Consider again the candidate address 0x80507619. The first bit position to be extracted is bit 0, which is on, so the next node becomes the right child. Bit 9 is then extracted, which again is on, so the next node is again the right child. Bit 17 is then extracted, which is again on, so the next node becomes the leaf node with QNet. The masking operation is then performed, and the address is deemed to belong to that class.

In the event that the masking operation fails, the algorithm then backtracks up the tree to find another mask that may apply. For example, if an attempt is made to match the address 0x80502680, the algorithm will zero in on the ZNet class, however, the XOR will signal that the address does not belong to this class. The algorithm will then backtrack up the tree, and an XOR with the mask for the HNet class will result in a match. Table 3.6 shows the results of the comparisons.

Sklower [36] notes that in the average case, the PATRICIA trees constructed from routing information are approximately balanced, which results in a search length of $1.44 \times log(N)$, where N is the number of entries. In the worst case, each bit in the address must be tested, which in the case of

Class	XOR Result	Mask
PNet	0000 0000 0101 0101 0010 0110 1000 0000	1111 1111 1111 1111 0000 0000 0000 0000
ZNet	0000 0000 0000 0000 0000 0010 1000 0000	1111 1111 1111 1111 1111 1111 0000 0000
HNet	0000 0000 0000 0000 0010 0110 1000 0000	1111 1111 1111 1111 0000 0000 0000 0000

Table 3.6: Binary results of XOR when backtracking required

IP addresses, would require 32 tests.

3.3.3 LC trie algorithm

The PATRICIA tree data structure uses *path compression* to compress sparsely populated sections of the tree by removing nodes with one-way branching. In contrast, an LC trie uses *level compression* for compressing parts of the tree that are densely populated [29]. The algorithm recursively compresses complete subtrees of height i into single nodes of degree 2^i. Nilsson and Karlsson discuss an efficient implementation of their routing algorithm that stores the entire routing table in an array.

One of the main differences in the lookup algorithm is that nodes can have different branching factors. Therefore, unlike in the PATRICIA tree algorith, where a single bit position is tested to determine the branch, in the LC trie algorithm, n bits need to be extracted for a branching value of 2^n. The value of these bits determines the branch to be taken. Each node stores information about its branching factor and a skip value. The pseudocode for this algorithm is shown below.

```
Current node ← Root
Position ← Skip
while ($n$ does not equal 0) do
  Extract $n$ bits starting at Position, store in Bits
  Branch ← value of Bits
  Current node ← Child number Branch
  Position ← Position + $n$ + Skip
end while
```

3.4 Concluding remarks

Reduced Instruction Set Computers present a viable alternative to the upward trend in processor complexity. By limiting the instruction set of the processor as well as the pipeline depth, savings can be achieved in terms of hardware cost and often execution time. The MIPS R2000 architecture is one of the original RISC implementations and provides an intuitive and simple instruction set. Its architecture is the platform on which experiments with the BSD routing code are conducted in this study.

Chapter 4

The Flexible Architecture Simulation Tool

The Flexible Architecture Simulation Tool enables rapid prototyping of new processor architectures. Developed by Onder and Gupta [31], FAST provides an Architecture Description Language (ADL)[1] that is used to describe both the processor behaviour and instruction set architecture (ISA). FAST provides an implementation of ADL that automatically generates a microarchitecture simulator, assembler, and disassembler. The simulator is cycle-accurate and executes approximately three orders of magnitude slower than the native architecture. FAST/ADL falls into the category of compiler-oriented ADLs and is most similar to EXPRESSION [16]. Pipelines are described explicitly as an ordering of functional units, and instructions are defined in terms of their opcode, operands, and the operations they perform in each pipeline stage they pass through. The ability to describe a machine at this level as opposed to at an RTL level has many advantages. An architect is freed from the laborious process of describing each register transfer and all the intricacies of the data and control paths. Furthermore, changes to the pipeline and instruction set architecture can be made much more easily at a high level, simplifying design-space exploration.

This chapter describes the FAST toolchain, its machine clock, and then provides an overview of the syntax and different constructs available in ADL. It concludes with a discussion on how FAST is useful today and where it should be extended in the future.

4.1 The FAST toolchain

A number of steps are required to build the FAST tools from an ADL processor description. In the first stage, the ADL compiler is used to transform an architecture description written in ADL into C++ source code. The generated Makefile is then used to compile this source code into the FAST simulation suite. A 6000 to 8000 line ADL architecture description is transformed into about 35000 lines of C++ code. This process is illustrated in Figure 4.1.

[1]From now on, unless otherwise stated, ADL will refer to the specific architecture description language of the FAST system.

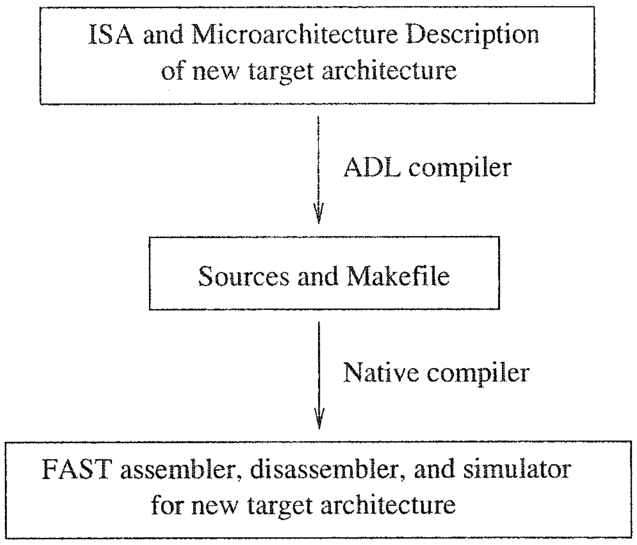
ISA and Microarchitecture Description of new target architecture
ADL compiler
Sources and Makefile
Native compiler
FAST assembler, disassembler, and simulator for new target architecture

Figure 4.1: Toolset generation

The FAST system consists of an assembler, a disassembler, and a simulator. The assembler is generated based on the ISA description in the ADL code, and is fully retargetable to any architecture and calling convention. The disassembler is not typically used standalone, but is normally called by the simulator when debugging mode is entered. The FAST simulator is cycle-accurate and bit-true, and explicitly models pipeline behaviour and instruction level parallelism. Once the three FAST programs are available, benchmark programs can be written, compiled, and executed on the simulated target architecture.

Although the FAST system can be targeted to any system call convention and assembly language format, it is most convenient to use those for which a C compiler is available, unless benchmark programs are to be written in assembly and custom libraries are available.

The first step toward running a benchmark program is to cross-compile it into assembly for the target architecture using the appropriate C compiler, if one is available. Once the assembly file is available, the FAST assembler is used to transform the assembly code into a binary file executable on the target architecture via the FAST simulator. The simulator is responsible for producing all of the performance data necessary for gauging the viability of the target architecture. This process is illustrated in Figure 4.2. FAST collects statistics about instruction usage, cycle counts, stall cycles, and memory accesses. It also provides functionality for collecting custom statistics from arbitrary locations in the data-path.

4.2 FAST machine clock

The operation of architectural components in FAST is described with respect to the FAST machine clock. Each *machine cycle* or *major cycle* of the machine clock consists of a number of *minor*

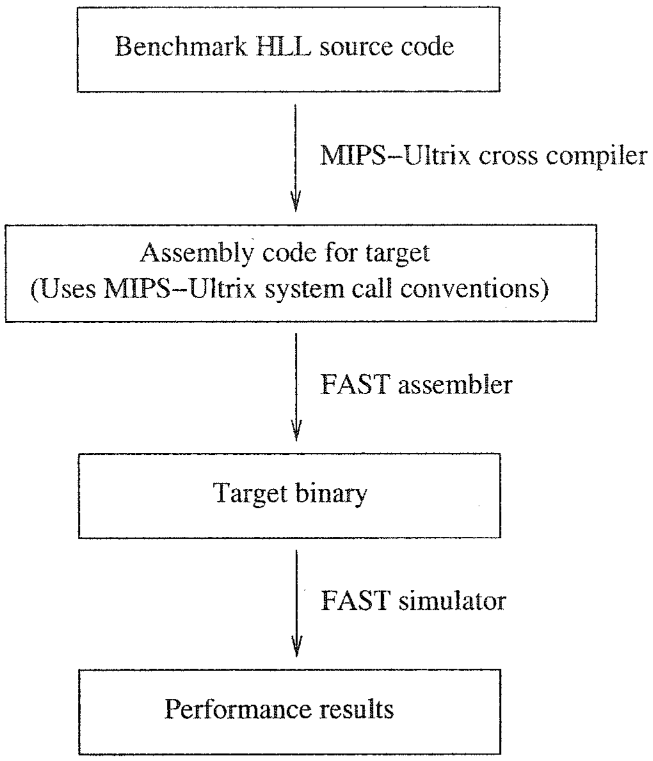
Benchmark HLL source code
MIPS–Ultrix cross compiler
Assembly code for target
(Uses MIPS–Ultrix system call conventions)
FAST assembler
Target binary
FAST simulator
Performance results

Figure 4.2: Generating and executing an object file

cycles, each of which is represented by a label, as shown in Figure 4.3. The first and last minor cycle comprising a major cycle are labeled *prologue* and *epilogue*, respectively, while all others in between are labeled as *intermissions*.

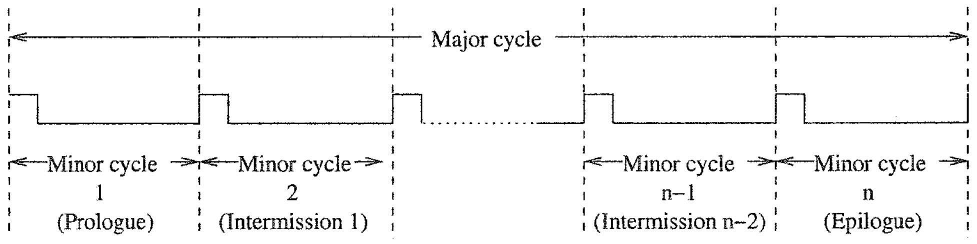

Figure 4.3: FAST processor cycles

In general, the prologue is responsible for receiving an instruction context from the previous pipeline stage, the intermissions operate upon the instruction context, and the epilogue sends the instruction context to the next pipeline stage.

Each action performed by an architectural component is annotated with the label of the minor cycle in which it should be executed. Actions common to a particular processing stage and minor cycle are grouped together to form *time annotated procedures* (TAPs). Given *n* minor cycles in a machine cycle, each processing stage can consist of up to *n* TAPs. Specific actions performed by machine instructions are annotated with the name of the processing stage in which they should be executed.

During each FAST machine cycle, both the actions associated with each component during that cycle and those of each active instruction annotated with the current cycle are executed.

4.3 Microarchitecture specification

Defining a processor architecture involves specifying *artifacts* and *stages*. Artifacts in ADL correspond to hardware objects with well-established operational semantics, while the functionality of stages exhibits more variation, and is often instruction specific.

4.3.1 Artifacts

ADL supports several types of artifacts as built-in types. In addition, the user can define their own custom artifacts. When declaring artifacts, attributes are supplied that determine the behavior of a specific implementation of the artifact. Artifacts built into the ADL language include *registers*, *register files*, *memory ports*, *caches*, *buffers*, and *latches*.

Registers and register files

A register declaration declares a simple register artifact. The only attribute associated with a register is its size in bits. For example, to declare a register named my_reg of size 32 bits, the following syntax would be used:

```
register my_reg 32;
```

The optional shadow keyword, placed at the beginning of the statement, renders the given register invisible to the instruction set.

A register file declaration declares an array of registers. Not surprisingly, it requires at least two attributes, those being the number of registers in the array and the size of each register. Register files can also be given the attribute shadow. An example shadow register file declaration is presented below.

```
shadow register file gpr[64,32];
```

This declares a shadow register file named gpr that has 64 entries of 32 bits each.

Register files have the quality that each entry in the register file can be assigned one or more aliases. This is taken care of in the same declaration along with the register file. For example, the following is used to declare a register file with 8 32-bit entries while assigning aliases for each of them:

```
register file mrf[8,32]
              $r0 0,
              $zero 0,
              $r1 1,
$r2 2,
$r3 3,
$r4 4,
$r5 5
$r6 6
$r7 7;
```

Following the initial declaration is a comma-separated list of aliases and indices. Aliases always begin with the $ character. An arbitrary number of aliases can be specified for a given register.

The following statements are both valid to access the 5th register in the register file:

```
mrf[4] = data_value;
$r4 = data_value;
```

Memory ports

Memory declarations define memory ports with given access latencies in units of machine cycles and and data path widths in units of bits. There are no optional attributes for memory declarations, and as such all such declarations have the following format:

```
memory memory1 latency 5 width 32;
memory memory2 latency 5 width 32;
```

The memory declaration declares a memory port, and not the memory itself. As such, the size of the memory cannot and need not be specified. It is assumed to be arbitrarily large to suit the needs of the program being executed on the target architecture. Multiple ports can be specified to the same memory.

Caches

The `cache` artifact is used to declare either instruction or data caches. Caches are *stackable*, meaning they can be used to construct memory hierarchies. Because of this, they need to be declared in relation to the memory object directly below them in the hierarchy. This is accomplished with the `of` clause. The general syntax of this looks like:

```
data cache dcache of memory1 512 4;
instruction cache icache of memory2 64 4;
```

A data cache of the first memory port is declared of size 512 kilobytes, and 4 words per line. Likewise, an instruction cache of size 64 kilobytes with 4 words per line is declared.

Custom artifacts

When the selection of built in artifacts does not suffice, ADL allows definition of custom artifacts with custom properties and behaviours. Artifacts are specified with the `artifact` keyword followed by a number of attributes. An artifact definition consists of a number of procedures that describe the behaviour of the artifact.

Definition of a custom artifact is best illustrated with an example. Figure 4.4 presents the ADL source code for a 2-bit branch prediction buffer. A more in-depth explanation of how a 2-bit predictor works is discussed in Section A.

Typically the first part of the artifact definition consists of data declarations. In this example, only one array is required, that which holds the two bits that determine the branch prediction. The size of the array is set at the value passed via the size attribute. This occurs on line 4.

When the simulation reaches the point where an artifact is declared the `initialization` procedure of the artifact is called. The purpose of this procedure is to perform any initializations on the artifact data. The `statistics` procedure is called at the end of the simulation. Any statements that are inserted into this procedure will be performed when the `<machine_name>.sim` file is being written. This is where `printf` statements should be inserted to output statistics about the artifact. This is accomplished with the `builtin` keyword. The `myself` keyword refers to the instance of the artifact being operated upon (similar to the `this` keyword in C++). The name assigned to the artifact can be printed this way.

The key to defining the behaviour of the artifact is via the `rvalue` and `lvalue` procedures. These procedures characterize how the artifact behaves when it is used as an r-value and l-value, respectively. An l-value consists of an expression that can appear on the left side of an assignment operator, whereas an r-value can appear on the right side of an assignment operator. In the case of the branch predictor artifact in Figure 4.4, when it is an r-value the prediction bits for the given address are returned. The `rvalue` procedure takes one argument, which corresponds to the index passed to the artifact instance elsewhere in the ADL code. For example:

```
predictor my_predictor size=256;
if (my_predictor[pc] == 0) then
   begin
      ...
   end;
```

Initial values for the attributes of an artifact are set by providing arguments during declaration of the artifact. The name of the attribute is specified followed by its initial value. Artifacts are accessed as r-values by passing an index value to the artifact, as in the previous example, where the return value is compared against zero. Values are returned from procedures by assigning a value to a variable with the same name as that of the procedure. In Figure 4.4, a value is assigned to `rvalue` at line 15.

Accessing an artifact as an l-value works in a similar manner, except that the procedure takes two arguments. The first argument corresponds to an index, the second to the r-value in the expression. Consider the following statements:

```
bitconstant TWO 1 0;
my_predictor[pc] = TWO;
```

```
artifact predictor attributes (size)
begin

   integer array predict_bits[size,2];   # Array of bits
   integer hash_val;                     # Calculated index value

   initialization        # Initialization actions
   begin
      forall predict_bits=0;
   end initialization;

   rvalue (addr)         # What to do if used as an rvalue
   begin
      hash_val = addr.[10:8];
      rvalue = predict_bits[hash_val];
   end rvalue;

   lvalue (addr, rval)  # What to do if used as an lvalue
   begin
      hash_val = addr.[10:8];
      if (rval == 0) then # Not taken
      begin
         if (predict_bits[hash_val] == ZERO) then
         begin
            predict_bits[hash_val] = ZERO;
         end
         else if (predict_bits[hash_val] == ONE) then
         begin
            predict_bits[hash_val] = ZERO;
         end
         else if (predict_bits[hash_val] == TWO) then
         begin
            predict_bits[hash_val] = ZERO;
         end
         else if (predict_bits[hash_val] == THREE) then
         begin
            predict_bits[hash_val] = TWO;
         end;
      end;

      if (rval == 1) then # Taken
      begin
         if (predict_bits[hash_val] == ZERO) then
         begin
            predict_bits[hash_val] = ONE;
         end
         else if (predict_bits[hash_val] == ONE) then
         begin
            predict_bits[hash_val] = THREE;
         end
         else if (predict_bits[hash_val] == TWO) then
         begin
            predict_bits[hash_val] = THREE;
         end
         else if (predict_bits[hash_val] == THREE) then
         begin
            predict_bits[hash_val] = THREE;
         end;
      end;
   end lvalue;

   statistics                      # Print statistics
   begin
      builtin printf(''Branch predictor %s\n'', myself.name);
   end statistics;

end predictor;
```

Figure 4.4: ADL source code for 2-bit predictor

The constant value TWO, which is the r-value, is assigned to the location in my_predictor to which pc indexes.

Aside from the basic procedures discussed above, an artifact definition can contain other procedures, separate pipelines, and other artifacts.

4.3.2 Pipelines

Defining the pipeline(s) associated with the microarchitecture involves first declaring the stage names followed by implementations of the TAPs that comprise each pipeline stage. Another critical component is to specify the data that are part of the *instruction context.*

Declaring pipelines and the instruction context

Pipelines are declared with the pipeline keyword. The declaration involves specifying a name for the pipeline followed by a list of its stages. For example, the standard MIPS pipeline can be declared in the following manner:

```
pipeline IPIPE (IF, ID, EX, MEM, WB);
```

As an instruction progresses through the pipeline, it carries data with it in a set of controldata registers. The set of these registers constitutes the instruction context, which stores data relating to the instruction's operands, destination register, intermediate results, etc. Each pipeline stage has the same type of context because the instruction context is the union of all the data required by all the stages in the pipeline. Declaration of the instruction context follows a pattern similar to other register declarations. This declaration is given by the controldata register keywords followed by a list of register names and sizes.

The controldata registers are accessible from within TAPs as well as from within the ISA specification. Both qualified and unqualified access modes are possible. In the case of unqualified access, the data accessed are that of the respective pipeline stage performing the access, or the stage associated with the label for the RTL segment in the semantic portion of the instruction declaration. The syntax is identical to that for normal register access. The syntax for qualified access of data involves specifying the name of the controldata element along with its respective pipeline stage. The general syntax looks like *controldata-element* [*stage-name*] .

Implementation of pipeline stages

Pipelines are implemented in ADL in a distributed fashion in the form of procedures. Machine cycles in FAST are divided into several minor cycles. These minor cycles include a *prologue*, a series of *intermissions* and an *epilogue*. Procedures must be specified for the prologue and epilogue of each stage, while those for intermissions are optional. The general format for defining procedures is as follows.

```
procedure STAGE_NAME ( prologue | intermission | epilogue )
begin
   STATEMENTS
      .
      .
      .
end STAGE_NAME
```

The statements that comprise the procedure define its behaviour with respect to the different instructions that it will process. Forwarding of instruction contexts, management of hazards, and other operations on instruction contexts are managed with a variety of keywords built into the ADL language.

Prior to its initial access, an instruction context must first be allocated using the `newcontext` keyword, which normally appears in the first pipeline stage. Once allocated, the instruction context can be operated upon. Failure to allocate a new instruction context for each instruction read from memory will stall the pipeline. Once an instruction context is no longer needed, it is deallocated using the `retire` keyword. Instruction contexts can be retired as either `stat` or `nostat`. This determines whether statistics about the instruction are recorded or not recorded, respectively.

Decoding of instructions is accomplished with the `decode` statement. The `decode` statement establishes a mapping from the instruction name to the current instruction context. The purpose of the `decode` statement is to load the attributes of the instruction into the `controldata` registers, provided the instruction context has been previously allocated. Once this has been accomplished, the `controldata` variables can be accessed by the pipeline stage.

The `send` statement is responsible for forwarding a particular instruction context from one stage to the next. Sends can be successful or unsuccessful. A successful send is achieved when the next stage is idle or performing a send in the same cycle. If a stage does not execute a send in a particular cycle, send operations of preceding stages will fail in that cycle, and those stages will repeat the send operation in the next cycle. All pipeline stages, with the exception of the final stage, must execute a `send` somewhere in their epilogue minor cycle, unless the instruction was retired prior to reaching the final pipeline stage.

The `send` keyword takes a single argument corresponding to the pipeline stage to which the current instruction context should be sent. This is important since instruction contexts can be sent to more than one pipeline stage. For example, a floating point add instruction may need to be forwarded to a different stage than an integer add instruction. This becomes especially apparent when the micro-architecture has more than one pipeline.

Stages may stall themselves through the use of the `stall` statement. The `stall` statement terminates processing of that stage for the remainder of the machine cycle. As a result, no `send` operation will be executed by that pipeline stage during that machine cycle. The `stall` statement has no effect on other stages. Stages further down the pipeline from that which stalled continue to execute their send operations. As a result, the stall statement has the effect of inserting a *bubble* into the pipeline.

In addition to stalling single pipeline stages, it is also possible to `freeze` and `unfreeze` the entire pipeline. The `freeze` statement functions in a manner somewhat opposite to that of the `stall` statement. While the `stall` statement will stall only the stage in which it was executed, the `freeze` statement will stall all stages except that in which it was executed. The only stage that may execute the `unfreeze` statement is that which executed the corresponding `freeze` statement. Freezing a pipeline might be useful in the event of a cache miss, for instance.

4.4 Instruction Set Architecture specification

Besides the architectural artifacts and pipeline stages themselves, the other key ingredient of a microarchitecture specification is that of the *instruction set architecture* (ISA). While the artifacts and stages define the general operations to be done on instruction contexts, the ISA specification defines specific operations for each instruction.

4.4.1 Instruction formats

Any given architecture places specific requirements on the format of its machine instructions. These instructions normally consist of a number of different fields that contain opcodes, operands, and other data.

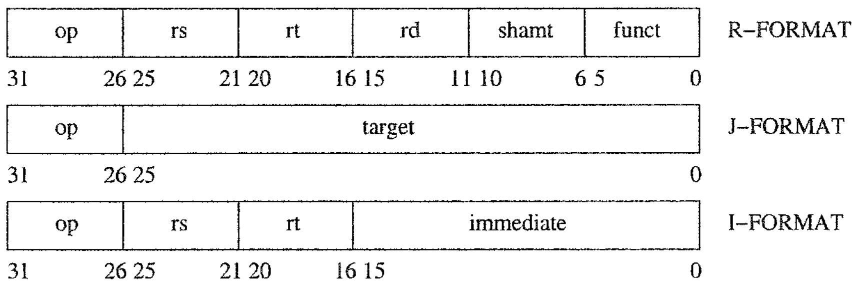

Figure 4.5: MIPS Instruction Formats

Consider the instruction formats for the MIPS architecture as shown in Figure 4.5. MIPS defines three instruction formats for integer operations, namely the *register*, *jump*, and *immediate* formats. Each instruction word is 32 bits in length, although the three formats consist of different fields with different lengths.

Instruction fields are defined in ADL by associating a *start bit* and *field width* pair with the name of the field. Instruction formats are not defined explicitly, but rather are defined implicitly as part of each instruction's binary part. The binary part of instructions are represented as a sequence of field expressions. This is simply an assignment of a value to a particular field of the instruction. This will be discussed in more detail in Section 4.4.3.

Declaration of the instruction fields is accomplished via the type keyword, followed by a list of field declarations. Each field declaration follows the following syntax.

```
FIELD_NAME FIELD_TYPE ( field | fixedfield ) START_BIT WIDTH
```

where FIELD_TYPE can be one of constant, integer, register, or signed. Constant fields are those that have values defined as constants elsewhere in the code. Register fields have values of machine register numbers. Integer and signed fields can contain values limited only by the size of the field in bits.

The difference between a field and a fixedfield has to do with when the value of the field is determined. Any instruction fields which are accessed before the instruction is decoded must be declared fixedfield.

An example declaration of the MIPS instruction formats presented in Figure 4.5 is shown below.

```
type
   op        constant field 31  6,
   rs        register field 25  5,
   rt        register field 20  5,
   rd        register field 15  5,
   shamt     integer  field 10  5,
   funct     integer  field  5  6,
   target    integer  field 25 26,
   immediate signed   field 15 16;
```

4.4.2 Attributes and opcode constants

Attributes in ADL perform the same function as enumerated data types in C++. Any given attribute can take on one of a specific set of values declared along with the attribute name. These attributes can then be used later to keep track of instruction types, operand types, and any other necessary conditions that must be checked to properly control instruction flow.

To declare an attribute is simply a matter of specifying an attribute name followed by a comma-separated list of possible values. The list is terminated with a semicolon. Multiple attributes can be declared with the same attribute keyword.

An example of declaring two attributes looks like this:

```
attributes
   i_class         : float_class,
                     integer_class,
                     branch_class,
                     long_integer_class;

   i_cycles        : single_cycle,
                     multiple_cycles;
end;
```

Opcode constants tell the assembler what binary form of an instruction to emit during assembly. These constants are used in the binary parts of the instruction definitions. An opcode constant consists of a name and a simple string of bits.

Declaring opcode constants begins with the bitconstant keyword, which is followed by a list of names and bit strings. The same type of declaration was shown in Section 4.3.1.

```
1    sll rd rt shamt
2       emit opcode=_special rs=0 rt rd shamt funct=_sll
3       attributes
4       (
5          i_class    : integer_class,
6          i_cycles   : single_cycle,
7          exu        : integer_unit,
8          c_what     : none,
9          dest_type  : integer_register,
10         lop_type   : none,
11         rop_type   : integer_register,
12         rop
13         i_type     : alu_type,
14         dest_reg   : rd
15      )
16      begin
17         exact s_EX
18            dest=(+rop) << (+shamt);
19         end;
20      end;
```

Figure 4.6: MIPS `sll` instruction declaration

4.4.3 Machine instructions

The instruction declarations are the heart of the ISA description and tell the assembler how to parse instructions and the simulator how to execute them. An instruction definition consists of a syntax part, binary part, and a semantic part. The syntax part tells the assembler how to parse the instruction. The binary part tells the assembler what machine code to emit, and indirectly describes the instruction formats of the architecture. The semantic part provides an implementation of the instruction with labeled register transfer level (LRTL) statements. Consider the MIPS `sll` instruction shown in Figure 4.6. The syntax part is found on line 1, the binary part on line 2, and the semantic part on lines 3 through 19.

The syntax part consists of the instruction mnemonic followed by any arguments the assembler should expect. In the case of the `sll` instruction, the assembler expects to find a destination register ID, a target register ID (which contains the operand), and a shift amount.

The `emit` keyword signifies the beginning of the binary part. What follows tells the assembler what to emit (hence the name) during the assembly stage. What follows the `emit` keyword is a list of instruction fields. The values of the fields may be those read by the assembler, they may be constants, or they can be values to which some transformation was applied by means of a procedure. The `_special` opcode is a constant that is defined along with other opcode constants in the manner described earlier. Since the `rs` field is unused in this instruction, it is assigned a value of zero. Following this are the `rt` and `rd` registers and the shift amount, all of which were read by the assembler, and finally the function value. In MIPS the function field is used to extend the number of instructions that can be implemented. The binary parts of the instruction definitions define the instruction formats for the architecture in question.

The third part of the instruction declaration is semantic part. First the instruction's attributes are set, and then the LRTL statements that define the behaviour of the instruction are provided. This section is delimited by `begin` and `end` keywords. RTL statements are grouped according to the pipeline stage in which they are to be executed. The `exact` keyword is used to specify the pipeline stage, and its respective RTL statements follow. For the `sll` instruction, a single operation is performed in the `s_EX` stage. The value of `rop` is loaded from the register file in the instruction decode stage. RTL statements can be specified for as many pipeline stages as required.

4.4.4 Macro instructions

ADL provides a means of declaring macro instructions, often called pseudo instructions. Such instructions are common in many compilers today such as `gcc`. Macro instructions normally break down into several machine instructions. This is the task of the assembler, and thus allows a slightly higher level of abstraction for the programmer and simplifies the assembly code.

The syntax part of macro instructions does not require fields from the instruction formats, but instead uses variables. It is not necessary to specify start bits or field widths for variables. They instead provide temporary storage for values read by the assembler. The three example variables declared below are used in the example macro instruction discussed next.

```
type
   rdest      register variable,
   rsrc1      register variable,
   rsrc2      register variable;
```

The syntax part of macro instructions looks the same as that for machine instructions. The difference is that all the arguments to the instruction must be variables. No instruction fields are allowed. Additionally, since macro instructions do not lead directly to binary code, there is no binary part in the macro definitions. Consider the following example:

```
sra rdest rsrc1 rsrc2 macro
   begin
      srav : rd=rdest rt=rsrc1 rs=rsrc2;
   end,
```

The syntax part of this macro instruction tells the assembler that whenever it sees the `sra` mnemonic that it should look for three values after it and load those into the variables `rdest`, `rsrc1` and `rsrc2`.

In the above example, only a single instruction is generated, that being the `srav` instruction. This instruction takes three arguments, and those are assigned the values in `rdest`, `rsrc1` and `rsrc2`, respectively.

4.5 Additional constructs

4.5.1 ADL Operators

Like any programming language, ADL contains a variety of operators used when defining control flow. These operators are presented in Table 4.1. All operators are binary, with the exception of the unary negation operator.

Operator	*Description*
`-a`	Negation
`a || b`	Concatenation
`a[b]`	Index
`[a:b]`	Bit group
`a.b`	Dot
`a:b`	Field width
`a |< b`	Bit extension
`a = b`	Assignment
`a == b`	Equality
`a < b`	Less than
`a > b`	Greater than
`a ^= b`	Not equal
`a << b`	Left shift
`a >> b`	Right shift
`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a % b`	Modulus
`a & b`	Bitwise AND
`a | b`	Bitwise OR

Table 4.1: ADL operators

A few of these operators are unusual enough that they deserve special mention. The bit extension operator is used to extend a single bit a over b spaces. It is frequently used with the concatenation operator. For example, the following statement concatenates a with bit b extended 8 bits, and assigns it to c:

```
c = a || (b |< 8);
```

The bit group operator is normally used with the dot operator to extract a sequence of bits from a data value. For example, the following statement extracts 8 bits from the 32 bit value c beginning from bit 15 and assigns it to location d:

```
d = c.[15:8];
```

The bit group operator is really just a specific use of the field width operator. In a more general sense, this operator is used to specify the width of a data value. It is sometimes used when specifying the width of an argument to a procedure.

4.5.2 Control structures

ADL provides two conditional statements that can be used to test the values of variables, registers, and attributes. The first is an `if-else` construct, while the second is a `case` statement that is analogous to a C `switch` statement. The `if-else` statement is written as follows:

```
if ( CONDITION ) then
   BLOCK
else
   BLOCK
```

The `else` clause is optional. A block consists of a `begin` keyword, followed by statements, followed by an `end` keyword. Note that if the block only contains a single statement, the `begin` and `end` keywords are not necessary.

The `case` statement can only be used with attributes. Essentially it operates in much the same way as a `switch` statement. The format looks like:

```
case ATTRIBUTE of
begin
   VALUE_1: STATEMENTS
   .
   .
   .
   VALUE_N: STATEMENTS
end
```

ADL also provides two looping control structures. The first is the `forall` statement that is used to initialize all the elements of a register file or integer array. Its use is demonstrated with the following example.

```
integer array my_array[1024,32];
forall my_array = 0;
```

An array of 1024 32-bit integers was declared, and then each integer was initialized to 0.

The second loop structure more closely resembles a traditional `for` loop. This example performs the same function as the previous, but using the `for` construct.

```
integer array my_array[1024,32];
for i = 0 step 1 until 1023 do
   my_array[i] = 0;
end;
```

The flexibility is somewhat more limited than with a C/C++ `for` loop, because a boolean condition is not checked at the end of each iteration. Instead only a counter is incremented.

4.6 Experiences with FAST

FAST has proven to be an extremely useful tool for design space exploration of processor architectures. In addition to the standard 5-stage MIPS architecture described in Chapter 3, ADL has been successfully applied to various speculative superscalar architectures based on the MIPS ISA

[30]. ADL is a fairly intuitive language for describing both structural and instruction set behaviour. At the present time, it is targeted mainly toward the functional description of standalone processors. Pipeline behaviour is described in an Algol-like language, and the operations performed by individual instructions are described at an RT-level. ADL excels particularly when it comes to modeling processor behaviour at a cycle-level. ADL models pipelining as a flow of instruction contexts through successive stages. Multi-cycle functional units are a natural consequence of this, and ADL allows the designer to specify an arbitrary number of such units and the interaction among them. Given ADL's explicit methods of dealing with resource conflicts, FAST presents itself as a very useful tool for identifying bottlenecks in both hardware and software. A designer can quickly identify sources of problems, and propose additional functional units or changes to the software architecture to deal with them. FAST also provides excellent statistics on instruction utilization, allowing the designer to quickly gauge the utility of the instruction set and make the appropriate additions, removals, and modifications.

There are a number of areas where ADL can be improved that would allow greater flexibility when describing SOC architectures. Although ADL can express multiple functional units in the same processor and allows for definition of coprocessors, scoping issues exist that make coprocessor definitions almost useless. In reality coprocessors should be able to communicate with one another. In ADL, however, coprocessors can only access those that were defined previously. As a result, two way communication is impossible. The only way to circumvent this problem is to model a coprocessor as a separate pipeline within the same processor. While this may be an acceptable short-term solution, ultimately ADL would benefit most from a resolution of the coprocessor scoping issues.

Another area in which ADL can be extended is to allow for more advanced caching. Currently, support only exists for simple direct-mapped caches. Although a programmer is free to construct whatever type of cache artifact he or she wishes, it would be helpful if more flexibility was available with the built in caching. This would require the ability to specify the set associativity of the cache, the replacement strategy, and whether it is write-through or write-back. Furthermore, with any extension of ADL allowing multiple autonomous processors, a method will be required for dealing with cache coherence.

One of the limitations of the FAST memory system is its single, infinite size memory. ADL provides constructs for memory ports, but not explicitly for memory. The memory size is limited only by that of the host machine, and there is no means to segment the memory into two or more autonomous regions. It would be much more useful if the programmer had the freedom to specify the size of the address space and if different memories could be declared for different purposes.

While ADL is retargetable to any architecture, current experiences have been limited to the MIPS ISA and system call conventions. An ADL description requires two sections where the programmer specifies simulator and assembler supplements in C++. The simulator supplements are responsible for setting up the initial conditions of the program, such as loading arguments onto the program stack, etc. The assembler supplements involve writing C++ functions to synthesize the various addressing modes of the processor. These functions are then used by the assembler when parsing the assembly code. In order to make ADL easily targetable to arbitrary architectures, it would help significantly if ADL constructs existed for handling these tasks. Specifically, the designer should be able to specify the addressing modes of the instruction set in ADL in a straightforward fashion, and the necessary assembler functions should be generated automatically.

Once the ability is in place to express multiple processors, ADL should be extended to include such structures as buses, switching fabrics, and I/O ports. This would greatly increase ADLs descriptive power in terms of SOCs and embedded systems. Such constructs would elevate FAST and ADL above a tool simply for modeling the behaviour of single processors. Much more complex systems such as microcontrollers, routers, switches, and other devices requiring outside communication could be simulated, and dataflow through entire embedded systems analyzed. Currently, some work is being done on extending ADL to model switching fabrics and routing coprocessors. With the inclusion of I/O ports, however, comes the need for hardware interrupts. Currently, FAST operates in a synchronous manner whereby events are triggered explicitly by the software or hardware, and the hardware has full knowledge of what will occur and when. Once asynchronous events are allowed, however, this changes, and the hardware should have a mechanism for detecting and responding to events.

What is further missing from ADL and FAST are methods of estimating cycle times, chip area, and power consumption. Since the language operates at an artifact level rather than a gate level, modifications to the architecture or instruction set have no effect on execution time beyond actual cycle counts. ADL would benefit from a mechanism that estimates the actual cycle time of the processor based on the complexity of the hardware and the length of the data path. This would allow a designer to more accurately judge the effects of structural changes on performance. Estimation of chip area would also be useful in many circumstances. This is of concern in many embedded systems, and reducing processor area frees up space for more memory, registers, cache, or additional functional units. Power consumption is also a critical performance measure, especially in wireless devices that run on batteries. In order to provide estimates of these performance metrics, FAST may need to be extended to perform some level of synthesis.

4.7 Concluding remarks

The Architecture Description Language provides powerful features for modeling processor behaviour at the cycle level. Pipelines are described explicitly as an ordering of stages, and the functions performed in each stage are grouped together by the processor minor cycle in which they occur. ADL also provides a number of built in constructs for describing memory hierarchies, and allows the programmer to design custom functional units as required. ADL provides a very precise method for describing the instruction set of the processor. Assembly syntax is defined in terms of instruction mnemonics and arguments, and the explicit definition of instruction formats leads to a natural method of describing the binary format of instructions. ADL currently suffers from some factors that limit its application to describing SOC architectures and embedded systems, and a number of these factors have been identified.

The FAST system provides an implementation of the ADL language and generates an assembler, disassembler, and cycle-accurate simulator from an ADL description. The simulator provides a variety of useful statistics about program execution, instruction usage, and resource usage, as well as on bottlenecks and conflicts in the data path. With the extensions described above, FAST and ADL could become one of the most sophisticated and useful hardware simulation systems available.

Chapter 5

Experiments and Discussion

Developing a hardware and software architecture for IP routing requires a multi-faceted approach. From the hardware angle, the problem can be approached in terms of the instruction set, pipeline architecture, and functional units in the processor. A major part of this thesis is to propose as small an instruction set as possible that does not sacrifice a great deal of performance. The expectation is that any minor increase in cycle count will be offset by the advantages of a smaller chip area and simplified microarchitecture. From the software standpoint, we approach the problem in terms of pipeline stalls and branch behaviour. Pipeline stalls bridge the gap between hardware and software, as they are dependent on both resource contention and data dependencies. Modifications to the hardware and software can reduce the number of stalls, which will increase performance. Branches play a role as well given the delays that they introduce. We show that a large number of branches are either always taken or never taken, which essentially amount to a wasted machine cycle. Software modifications can eliminate these. This chapter begins with a discussion of our experimental configuration, and continues with the results of these three approaches and their implications. The chapter concludes by tying these results together.

5.1 Experimental configuration

The Flexible Architecture Simulation Tool is the basis for the experiments and results discussed in this chapter. Using an ADL description of the R2000 architecture, two types of IP routing algorithms were tested. Figure 5.1 displays a block diagram of the test arrangement.

5.1.1 ADL Implementation of MIPS R2000

The ADL architectural description used in this study is the standard five-stage MIPS pipeline with internal data forwarding. The key features of the processor are implemented:

- *Registers.* All thirty-two 32-bit integer registers are present.
- *Pipeline.* The five stage pipeline architecture is fully implemented and conforms to the specification described in Chapter 3. Internal data forwarding exists so that operands are available

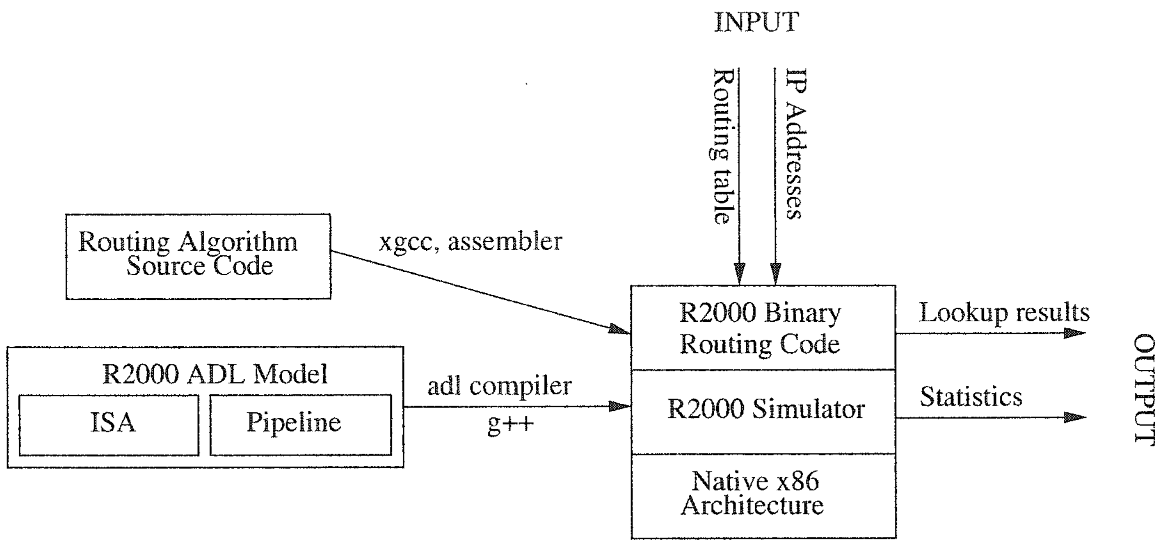

Figure 5.1: Experimental configuration

before they are written back to the register file. Data dependencies are checked in the decode stage and the pipeline stalls when necessary.

- *Instruction set.* The full MIPS integer instruction set is implemented in the ISA description. All three instruction formats are provided, and all of the MIPS addressing modes are available.
- *Floating point unit.* Although not implemented as a separate coprocessor, the ADL description includes an implementation of a floating point pipeline. A subset of the R2010 floating point instruction set is provided, as well as 16 double-precision floating point registers.

These features provide the functionality necessary to accurately simulate the execution of MIPS object code. The following features of the R2000 were not implemented for the reasons noted:

- *Caching.* The R2000 provides on-chip cache control logic that allows separate data and instruction caches. The simulation assumes zero memory latency, and therefore caching would not serve any useful purpose. The implications of this are discussed later.
- *Virtual memory.* The R2000 provides a Translation Lookaside Buffer (TLB) that maps virtual addresses to the physical address space. This is not required in ADL because the simulated physical memory is a full 32-bit space.
- *System control coprocessor, CP0.* CP0 supports the TLB and system calls. Although system calls are considered exceptions in MIPS, they are handled by the main pipeline. Other exceptions are either ignored, cause the simulation to terminate, or open the debugger. Thus, CP0 is not required in this work.

The ADL code that implements the R2000 pipeline architecture is presented in Appendix B.

5.1.2 Routing algorithms

The basis for our measurements is the radix-tree routing algorithm from the 4.3 Reno release of Berkeley UNIX as described in Chapter 3. The results obtained from this algorithm are then compared to those from an implementation of the LC Trie algorithm of Nilsson and Karlsson [29].

5.1.3 Routing tables

The routing tables used as input to the BSD radix tree and to the LC trie routing algorithms are taken from Internet core routers. The tables were provided by the Internet Performance Measurement and Analysis Project [23]. Each entry in the tables consists of an address and a mask. The table used for the performance measurements that follow contains 2000 entries. In the early stages of our work, larger tables of up to 20000 entries were used. Although absolute instruction usage counts were much larger, the relative percentages for different table sizes were almost identical. For this reason, the majority of the measurements were conducted with a smaller table for shorter simulation run times.

5.1.4 Driver program and IP address generator

The driver program interfaces with the various routing libraries, reads the routing table and generates IP addresses for lookup. It first constructs the routing table from the input file. Then the sequence of addresses to be looked up in the table are generated by repeatedly choosing four random values between 0 and 255 and concatenating the resulting bytes to form a 32-bit IP address.

The driver program is set to generate 8192 lookups. Larger and smaller numbers were tested, however, as with different routing table sizes, the relative percentages of the instructions executed remained nearly identical. For this reason, sequences of 8192 address lookups is the basis for all the measurements discussed in this chapter.

The routing code and the required libraries are cross compiled with `gcc` version 2.7.2 into MIPS-Ultrix assembly format. The command used is

```
xgcc -S -D__builtin_va_list=void* -O3 *.c
```

The `-S` option instructs the compiler to leave the code in assembly format. The `-D` option tells the preprocessor to create the macro given as the argument. In this case, the type `__builtin_va_list` is replaced with `void*` during preprocessing. Finally, the `-O3` option instructs the compiler to use the highest level of optimization.

The routing code makes use of a number of libraries. These include `stdlib.h, string.h, sys/param.h, sys/systm.h, sys/malloc.h,` and `sys/syslog.h`. The statistics reported about instruction usage during routing table lookup include those instructions required by any function calls in these libraries.

5.2 Instruction set optimization

Optimization of the instruction set involves the application of one or more design metrics to the statistics generated by the FAST simulator. The result is a simpler instruction set that eliminates some of the less frequently-used instructions, while keeping those that are most important.

5.2.1 Design metrics used in this thesis

Holmer discusses a variety of metrics used in instruction set design. In this study, there are three that are of primary importance. These are instruction count, clock cycles per instruction, and clock cycle time. Of these, only instruction count can be measured directly. Reducing CPU time is the main objective of our work. CPU time is a function of the instruction count of the program, the average number of clock cycles per instruction and clock cycle time, as shown in Equation 5.1:

$$CPUtime = I \times CPI \times T \tag{5.1}$$

where I is the number of instructions executed, CPI is the number of cycles per instruction, and T is the clock cycle time. CPU time can be reduced by reducing any of these three terms. Of these three, I is easily, directly and accurately measured with FAST. Moreover, it is easy to determine the frequency with which individual instructions are used. Cycles per instruction (CPI) is of less interest to us because with a fixed pipeline, CPI remains relatively constant. Assuming the MIPS pipeline remains full throughout the execution of the program, integer arithmetic and logic instructions can be retired every clock cycle, so CPI is 1. In fact, stalls due to data dependencies increase this value to about 1.6 (assuming zero cycle memory latency). By reducing the number of stalls, this value can be lowered to be closer to 1. FAST provides detailed information about both CPI and the number of stall cycles by which it is affected. Clock cycle time (T) is related to such factors as VLSI chip area and the critical path length through the processor. FAST does not provide estimates of clock cycle time because ADL models processors at an artifact level rather than at a gate level. Changes to the architecture, therefore, have no effect on cycle time. For this reason, it is also difficult to estimate overall execution time. ADL does give us accurate information about instruction count and CPI.

The goal, therefore, will be to minimize instruction count and CPI. Doing this naively, however, will enlarge rather than reduce the size of the instruction set. Much of the related work has focused on generating application-specific instructions to reduce code size and to make better use of the hardware. The goal of this thesis is to make the ISA as small as possible without sacrificing a great deal of performance, with the aim that any increase in instruction count will be offset by a larger decrease in CPI as well as increased hardware efficiency.

The 1% rule

In the late 1970s, instruction set design changed from being an art based entirely on human intuition into an iterative engineering process [20]. Advances in retargetable compilers allowed benchmark programs to be compiled for a new architecture and simulated, providing insight into the utility of the ISA and beneficial changes to it. This was part of the RISC design process used by the MIPS company when designing their first processor [24]. "Any instruction added for performance reasons had to provide a verifiable 1% performance gain over a range of applications or else the instruction was rejected." The 1% rule can be stated more concisely as:

$$\Delta I + 100{\times}\Delta C/C < 0 \tag{5.2}$$

where I is the number of unique opcodes, and C the number of machine cycles required to execute the benchmark. The addition of one instruction equates to $\Delta I = 1$, which must be offset by at least a 1% drop in cycle count to be acceptable.

Human ingenuity is still required to propose instructions to be added to the ISA. The 1% rule is only a criterion for filtering trial instructions, or in the case of this study, existing instructions.

5.2.2 Instruction substitution

In some cases, instructions are not used at all and can be safely removed from the instruction set. In other cases, however, seldom-used instructions can be removed, but their functionality must be preserved. In this case, a machine instruction can be substituted with a pseudo instruction or assembler macro. The macro can then be expanded to a sequence of one or more machine instructions that replicate the functionality of the original instruction.

As a simple example, consider a `move` machine instruction in a hypothetical instruction set that moves a value from one memory location to another. Such an instruction may have the form "`move $5,$6`", where `$5` contains the target address, and `$6` the source address. Such an operation could be replaced with the following sequence of instructions:

```
load  $temp,$6
store $temp,$5
```

In the event that a macro generates more than one machine instruction, a temporary register is normally required to store intermediate values. In MIPS this register is `$1`.

There are a number of issues that one must be aware of when substituting machine instructions with assembler macros. First and foremost, if the number of instructions required by the macro is too many, and the machine instruction is used too often, the impact on performance may be unacceptable. A balance must be struck between hardware complexity and execution time. Second, the compiler should ideally be made aware of any changes so that it can correctly generate assembly code. This is especially important for an optimizing compiler that inserts instructions into delay

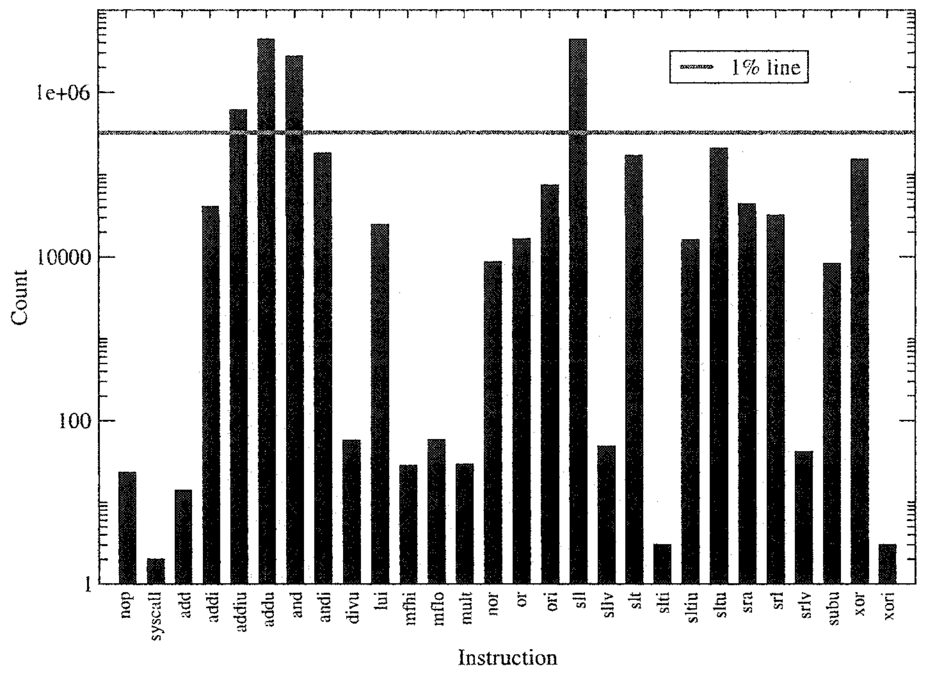

Figure 5.2: Arithmetic/logic instruction counts for radix tree code with original ISA

slots. If an instruction in a delay slot is a pseudo instruction, and is expanded into more than one instruction by the assembler, only part of the operation will be completed if the branch is taken. The solution to this is not to permit the compiler to insert any pseudo instructions into delay slots if they generate more than one machine instruction.

5.2.3 Initial results

This section presents the initial results from the unmodified instruction set. The values represent the number of instructions executed for the routing table lookup only, and do not take into account table construction. Construction of the routing table is a different matter and can be dealt with separately. Routing table construction is not a key element of router performance as updates occur infrequently. Furthermore, these functions are often handled by a unit that is separate from the data forwarding plane of the router. Figure 5.2 displays the execution frequencies for integer arithmetic and logic instructions, and Figure 5.3 the frequencies for branch, load, and store instructions.

Both graphs include lines that correspond to 1% of the total number of instructions executed from all categories. This illustrates the 1% rule graphically. Although the 1% rule is meant as a measure of overall performance, it can be applied strictly to instruction counts if CPI and clock cycle time do not change. Recall from Equation 5.1 how execution time is a product of these three quantities. Since FAST provides no data about clock cycle time, it is assumed to be constant. Furthermore, changes to the instruction set have very little effect on CPI, so this is assumed to be constant as well. Instruction count is therefore the only remaining variable.

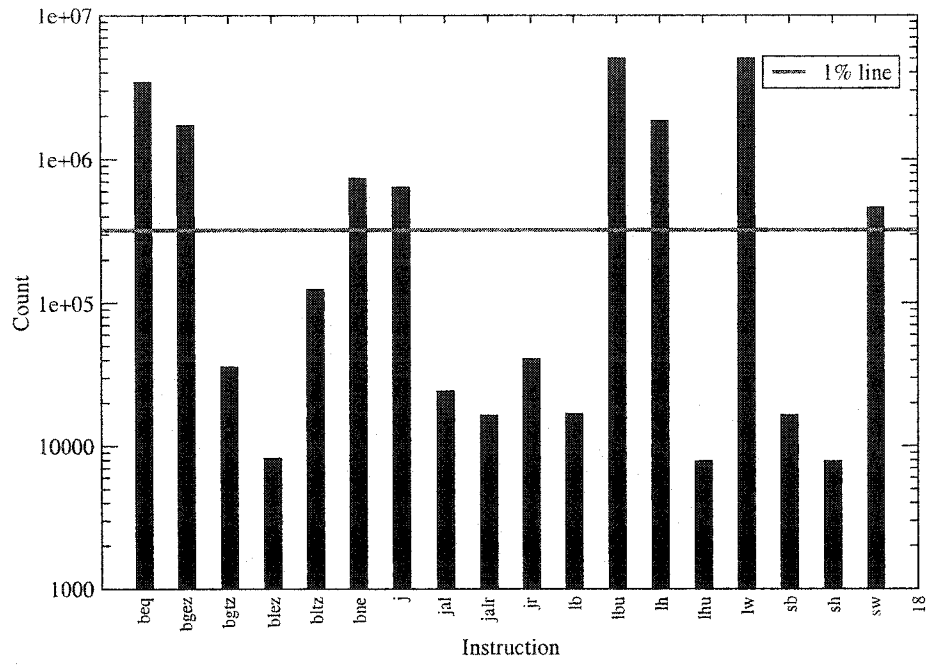

Figure 5.3: Branch, load, and store instruction counts for radix tree code with original ISA

Clearly, the majority of the arithmetic and logic instructions are candidates for removal. There are only four instructions that pass the 1% rule, and a few more come close. A significant number of instructions have counts of less than 100, and a few (which are not displayed) are not used at all. The most commonly used instructions are integer *add*, *logical AND*, and *left shift*. The *set on less than* instruction, used for checking conditions, is also heavily-used, as is *exclusive OR*. These elementary instructions should therefore form the basis of the routing ISA. The remaining instructions will be examined in more detail later.

Of the branch and memory access instructions, a few more pass the 1% rule. Some of the branch instructions and load/store instructions are candidates for substitution, while those that are not used at all can be removed completely.

The average number of instructions executed per address lookup is 3933. If stall cycles are considered, the average number of machine cycles required per lookup is 6747, which means over 58% of all machine cycles are stalls. This is contrary to popular belief that at most 150 instructions are required per lookup. As will be shown in Section 5.2.6, even a modern algorithm requires more than 150 instructions per lookup.

Section 5.3 discusses the reasons for the stalls and suggests some optimizations.

5.2.4 Unused instructions

The first step toward streamlining the MIPS ISA is to determine which instructions go unused completely. These instructions can be safely removed from the instruction set without worry. Given the nature of the FAST toolchain, if an instruction was removed that was generated by the compiler, the assembler would detect this and issue an error. If no error is issued, it means that the instructions that are removed are not used anywhere in the benchmark code.

Although FAST does not explicitly provide details about instructions that are unused, they are easy to determine by comparing the statistics generated to the entire list of opcodes. It turns out that relatively few instructions go completely unused. These are two branch instructions, five load/store instructions, and six arithmetic/logic instructions. Table 5.1 summarizes these instructions.

bgezal	bltzal
div	lwl
lwr	lwu
mthi	mtlo
multu	sub
swl	swr
srav	

Table 5.1: Unused MIPS instructions

The compiler uses the unconditional *jump and link* and leaves the conditional *branch and link* instructions unused. Many of these are unconditional function calls. Otherwise, the decision of whether to call the function is made by an earlier branch instruction. While the *divide* and *multiply unsigned* instructions go unused, the *divide unsigned* and *multiply* instructions are used very rarely. The assembly code contains no instances of the `divu` instruction, which means it must be used in one of the library functions. The assembly code contains a single instance of the `mult` instruction, although it is part of the driver program and not the actual routing code. The `mthi` and `mtlo` instructions are used to move arguments to the LO and HI registers, which store the results of multiply and divide instructions. Their counterparts are the `mfhi` and `mflo` instructions, which could be eliminated if the need for division was removed. The instructions `lwl` and `lwr` are somewhat specialized and are used to reference specific bytes in memory and place them in the left or right portion of a register, respectively. The `swl` and `swr` perform a similar operation, except they move bytes from a register to memory.

5.2.5 Examination of used instructions

The results from section 5.2.3 highlight a variety of instructions that are candidates for substitution. Some of these are used very infrequently, or are only used by library functions and not the routing code itself. The following discusses each instruction in detail, and where possible and desirable, how it can be substituted with a pseudo instruction. Where substitutions are suggested, they are presented in the format of ADL macro instructions. Arithmetic instructions are discussed first, followed by logic, branch, and load/store instructions.

add

Contrary to initial expectations, the add instruction is never generated by the compiler. The routing code utilizes only unsigned 32-bit addition, which makes sense in the context of IP address manipulation. The add instruction is also never generated by assembler pseudo instructions. The low execution frequency of this instruction suggests that is is used a limited number of times in one or more library functions. Its very low frequency also makes it a candidate for substitution.

The MIPS architecture makes no distinction between signed and unsigned arithmetic. The only functional difference is that signed arithmetic can cause an overflow exception, while unsigned cannot. As a result, it is easy to implement signed addition with unsigned addition. If the overflow check needs to be enforced, this can easily be done in software. The new add macro instruction is shown below.

```
add rdest rsrc1 rsrc2 macro
   begin
      addu : rd=rdest rs=rsrc1 rt=rsrc2;
   end;
```

The syntax for macro instructions consists of an assembly syntax specification followed by one or more instructions to be generated. The assembly syntax consists of the opcode name followed by a list of arguments. In the case of the add macro, a destination register and two source registers are required. These arguments are specified using variables defined earlier. The variables can then be manipulated and are eventually assigned to the fields of the instructions to be generated.

Substitution of add requires only a single instruction to be generated, so there is no increase in cycle count over the original instruction. It may be possible to modify the libraries directly so that add is no longer required, because this instruction is never used by the routing code but only in library functions,. If this is done, then the instruction can be eliminated completely and an assembler pseudo instruction is not required.

addi

Similar to the add instruction, the addi instruction is never generated by the compiler, although it is generated by the assembler in two pseudo instructions: subu and la (*load address*). These pseudo instructions must therefore be modified if the addi instruction is to be removed. Substituting this instruction with a macro is as easy as the previous case. There are actually two choices. The first will generate only a single instruction, while the second will generate two. Both are presented and discussed.

```
addi rdest rsrc1 src2 macro
   begin
      addiu : rd=rdest rs=rsrc1 uimmediate=src2;
   end;
```

In this case, the arguments to addi are simply assigned to their counterparts from addiu. This will be effective if the addiu instruction is retained. In the event that it is not, the following macro can be used.

```
addi rdest rsrc1 src2 macro
   begin
      ori   : rt=1 rs=0 immediate=src2;
      addu  : rd=rdest rs=rsrc1 rt=1;
   end;
```

The `ori` instruction is used to load the immediate value into a temporary register. That register is then used as an argument to the addu instruction.

addiu

Once again, for the BSD routing code, the `addiu` instruction is never generated directly by the compiler, but only by the assembler in pseudo instructions. The decision as to whether to substitute this instruction with a macro is more difficult. If one wishes as small of an instruction set as possible, it can be done. The instruction passes the 1% rule, however, and the required macro instruction generates two machine instructions. This may have an unacceptable performance impact.

This instruction is also used by the addu macro that takes an immediate argument, and by the `li` macro. As a result, substitution of this instruction can have a performance impact on those operations as well.

addu

The addu instruction is used heavily by the compiler, and is one of the most-used instructions in all of the BSD routing code. It is also generated by the addu macro (immediate format) and the move macro. Considering the very high frequency of this instruction, it should not be substituted.

lui

The `lui` instruction is never generated directly by the compiler, however it is used in several macro instructions. Its replacement macro instruction generates two instructions.

```
lui rsrc1 src2
   begin
      ori : rt=1 rs=0 immediate=src2;
      sll : rd=rsrc1 rt=1 shamt=16;
   end;
```

Once again, `ori` is used to load the immediate value into a register. That value is then shifted left by 16 bits, which is equivalent to the functionality of `lui`.

sll

This instruction is generated by the compiler and is often used as a substitute for the nop instruction. It is one of the most heavily-used instructions, and one of the few that pass the 1% rule. While it could be substituted, the performance consequences would be very bad.

sllv

This instruction is another that is never generated when compiling the routing code. It is used by an `sll` macro that takes an immediate argument, but this instruction is never generated by the compiler either. The limited frequency of `sllv` suggests that it is used in library functions. There is no obvious way of substituting this instruction with a macro. Unless the relevant library functions can be modified to no longer require it, the instruction must remain in the instruction set.

slt, sltu

Both of these instructions are used heavily by the compiler and are used by a variety of branch macros. While neither pass the 1% rule, they both come close, and provide a vital functionality for comparing values in registers. They cannot be removed.

slti, sltiu

In contrast to the previous two instructions, these are never generated by the complier, and the only macro that uses the `sltiu` instruction is an `sltu` macro. These instructions, in particular `slti`, are also less frequently used. They can be substituted in the following ways.

```
slti rsrc1 rsrc2 src3 macro
   begin
      ori : rt=1 rs=0 immediate=src3;
      slt : rd=rsrc1 rs=rsrc2 rt=1;
   end,

sltiu rsrc1 rsrc2 src3 macro
   begin
      ori  : rt=1 rs=0 immediate=src3;
      sltu : rd=rsrc1 rs=rsrc2 rt=1;
   end;
```

The idea with both is similar to earlier instructions with immediate arguments. The goal is to load the immediate value into a temporary register, and then proceed with the corresponding three register instruction.

sra, srl, srlv

Of these three instructions, only `sra` is used by the compiler. The others are not used by the compiler or assembler at all. Removing them would require modification to the library functions that use them. There are no obvious ways of substituting these instructions with macros.

subu

While it is possible to implement subtraction in terms of addition, it is not efficient. The right operand must first be negated, which involves an exclusive OR against a string of ones, which must first be loaded into a register. The result must then be incremented to form the twos-complement of the right operand. The result is then added to the left operand. Overall, up to six instructions are required to perform these operations. Therefore, it is probably more efficient to keep the hardware implementation of the subtraction instruction.

and, andi

The and instruction is used heavily by the compiler and is one of the fundamental operations that are required by the architecture. Removing it is not possible or desirable. The andi instruction, however, is much less used and can easily be removed. The code for the macro is shown below.

```
andi rsrc1 rsrc2 src3 macro
   begin
      ori : rt=1 rs=0 immediate=src3;
      and : rd=rsrc1 rs=rsrc2 rt=1;
   end;
```

nor

The nor instruction is rarely used by the compiler and is never used by any macro instructions. Its operation is easily implemented in terms of two other instructions.

```
nor rdest rsrc2 rsrc3 macro
   begin
      ori : rt=1 rs=0 immediate=0xffffffff;
      or  : rd=rdest rs=rsrc2 rt=rsrc3;
      xor : rd=rdest rs=rdest rt=1;
   end;
```

The first thing required is a string of 1s, which are loaded into the temporary register. The or operation is then performed on the input values and stored in the destination register. The xor of the result and the 1 bits is then calculated to invert the bits, and the result again stored in the destination register. This macro can be simplified by using the xori instruction, however, the goal is to remove that instruction as well.

or, ori

Both of these instructions are necessary and are used by the compiler. While it is possible to substitute ori in similar ways as other immediate instructions, all of these instructions require ori for loading immediate values into registers. Although this can also be done with addu, ori is already frequently used and can be of benefit if left in the instruction set.

xor, xori

Although the frequency of xor falls well short of the 1% rule, it is one of the fundamental operations that should be preserved in the instruction set. The PATRICIA tree algorithm relies heavily on this operation for performing lookup, so removing it would not be a good decision. The extremely low use of xori makes it an excellent candidate for removal. It can be implemented with xor and ori in the manner shown below.

```
xori rdest rsrc2 src3 macro
   begin
      ori : rt=1 rs=0 immediate=src3;
      xor : rd=rdest rs=rsrc2 rt=1;
   end;
```

Conditional branches

The most heavily used conditional branch instruction is beq. The others fall significantly short, and the functionality of those other than bne can be replicated with a macro that produces either two or three instructions.

```
bgez rsrc1 address macro
   begin
      slt    : rd=1 rs=rsrc1 rt=0;
      beq__  : rs=0 rt=1 b_offset=<address.delta.jump_address>;
   end,

bgtz rsrc1 address macro
   begin
      ori    : rt=1 rs=0 immediate=1;
      slt    : rd=1 rs=rsrc1 rt=1;
      beq__  : rs=0 rt=1 b_offset=<address.delta.jump_address>;
   end,

blez rsrc1 address macro
   begin
      ori    : rt=1 rs=0 immediate=1;
      slt    : rd=1 rs=rsrc1 rt=1;
      bne__  : rs=0 rt=1 b_offset=<address.delta.jump_address>;
   end,

bltz rsrc1 address macro
   begin
      slt    : rd=1 rs=rsrc1 rt=0;
      bne__  : rs=0 rt=1 b_offset=<address.delta.jump_address>;
   end;
```

The bgez and bltz macros can take advantage of register $0, which always contains a value of zero. The slt instruction can then be used to compare a value to 0. There is no predefined register with the contents 1, therefore this must first be loaded as an immediate value into a temporary register for the bgtz and blez macros. An alternative is to keep the slti instruction.

Unconditional branches

The MIPS ISA contains four unconditional branch instructions. Each performs a different function, and none are easy to remove. The only way that any of these instructions could be removed from the architecture is if the routing code or the compiler is modified. Branches will be discussed in more detail in Section 5.4.

Loads and stores

The BSD routing code makes heavy use of three load instructions that load data as bytes, halfwords, and words. Given the nature of the application, loads and stores should be optimized, because a great deal of memory access is required for IP address lookup. In total, eight load and store instructions are used. Reducing this number is more likely a compiler problem than an architectural one. From an architectural standpoint, there is little that can be done to eliminate any of these, given the unique functions they perform.

5.2.6 Proposed instruction set

Once the substitutions discussed above are implemented, what remains is the reduced MIPS instruction set presented in Table 5.2.

Arithmetic/Logic Instructions	**Branch Instructions**
addu	beq
and	bne
or	j
ori	jr
sll	jal
sllv	jalr
slt	
sltu	**Load/Store Instructions**
sra	lb
srl	lbu
srlv	lh
xor	lhu
subu	lw
addiu	sb
	sh
Special Instructions	sw
syscall	

Table 5.2: Reduced MIPS instruction set

The proposed instruction set contains 29 instructions out of the original 58 (not including coprocessor instructions). This is a 50% reduction in the instruction set size. Without examining compiler behaviour and performing a more detailed analysis of the algorithms in the BSD routing code, it will be difficult to reduce the instruction set further, because the operations in this instruction set are mostly independent of one another. The inability to reduce the load/store instruction set beyond the instructions that are unused is the least satisfying aspect of this instruction set.

In order to judge the effects of the modified instruction set, the routing code was run with the same configuration on the modified architecture. Overall, the number of instructions required per lookup increased from 3933 to 4197, an increase of roughly 7%. Figures 5.4 and 5.5 show the frequencies for the instructions in the new instruction set. As expected, the counts that have increased correspond to those instructions that are generated by the new assembler macros in place of the instructions that were removed. The most significant increase is observed in the `slt` instruction due to its use in four branch macros. The only instructions whose counts changed in Figure 5.5 are `beq` and `bne`. The total number of instructions affected is 11.

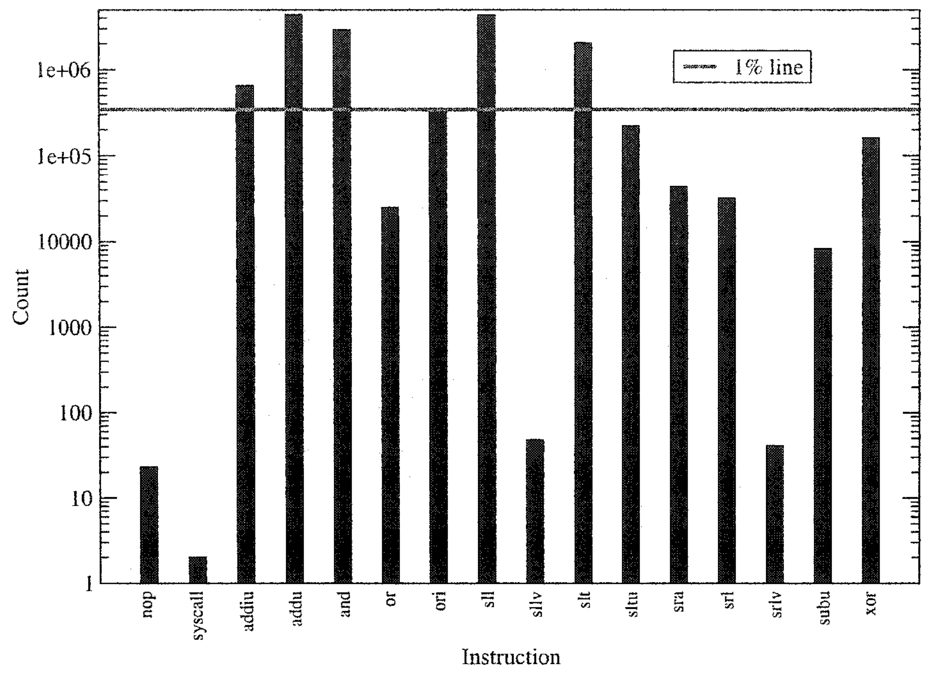

Figure 5.4: Arithmetic/logic instruction counts for radix tree code with modified ISA

A variation of this instruction set is to keep the four branch instructions that were substituted with macros. The rationale behind this is explained in Section 5.3. This version of the instruction set was used to test for variations in the instruction counts with different IP address traces. In total, five different IP address traces were used. The maximum total variation in instruction count observed is less than 0.15%, with the mean variation being approximately 0.06%.

Comparison case: the LC trie algorithm

The goal of a comparison is to determine whether the modified instruction set is suitable for more than one algorithm. The modified instruction set appears to be compatible with the LC trie algorithm. Multiplication and division are once again not used by the algorithm itself but only by the driver program and library functions in very small amounts.

Figure 5.6 shows the combined statistics for all instructions used during routing with the LC trie algorithm. The major difference in the statistics is that the LC trie approach requires only 277 instructions per lookup on average, a better than ten-fold reduction over the radix tree algorithm. The LC trie also has a much better overall utilization of the instruction set. Many more instructions pass the 1% rule, and most of those that do not come close. Furthermore, the number of unique load and store instructions required is smaller, and the overall number of memory accesses is reduced.

When run on the original R2000 architecture, the LC trie algorithm requires 277 instructions per lookup. On the modified instruction set, each lookup requires 288. The increase is just under 4%, which is better than the radix tree algorithm.

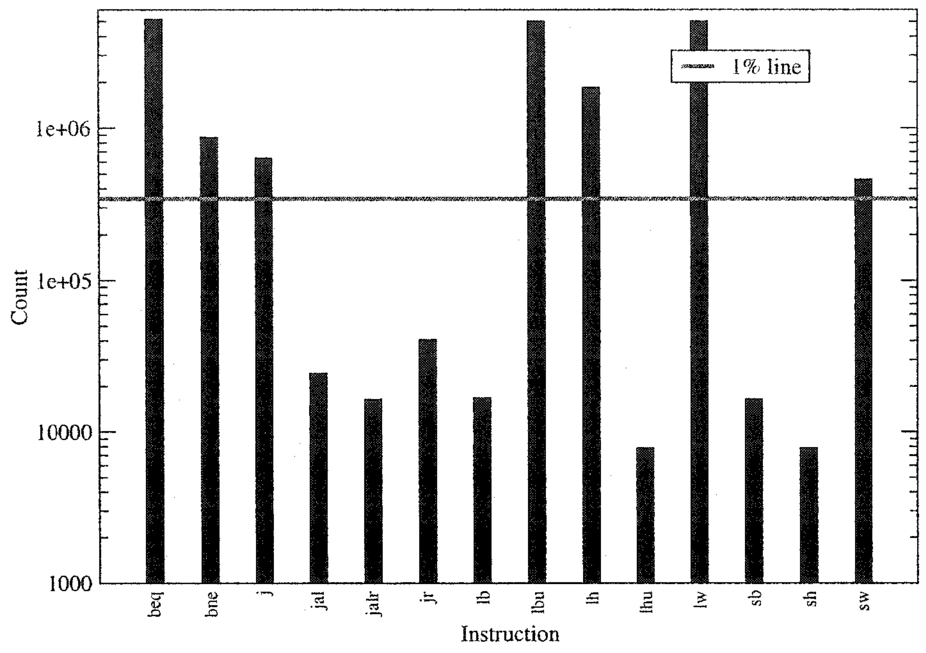

Figure 5.5: Branch, load, and store instruction counts for radix tree code with modified ISA

Comparison to IBM instruction set

The study by IBM discussed in Section 2.2.6 arrived at a general instruction set for routing data within multiprocessors. This instruction set supports a variety of routing algorithms for various interconnection network topologies. In total, the IBM instruction set contains sixteen instructions in six categories. One of the differences between the IBM instruction set and our modified MIPS instruction set is the former's limited memory access instructions. In particular, it does not provide any means of storing data to memory, and only very limited capability of loading values from memory. These operations are mainly provided for increasing the flexibility of the architecture by allowing it to run more complex algorithms. The investigation is mainly centred around oblivious routing schemes whose algorithms can determine how to route a packet based on its tag and the current node alone. For such schemes, table lookup is not necessary, and therefore support for such functions is very limited.

A second difference between the two instruction sets is their size. While the IBM instruction set is required to support only a very restrictive set of operations, our modified MIPS instruction set must still provide some level of support for more general programs, algorithms, and operating system functions. The ALU operations from both instruction sets are quite similar in the operations they provide, although ours are somewhat more flexible. Both instruction sets provide addition and subtraction operations as well as logical AND and XOR instructions. The `cmp` instruction of the IBM instruction set is similar to the `slt` and `sltu` instructions in ours. Both instruction sets provide left and right shift instructions, although ours also allows values to be shifted by a variable

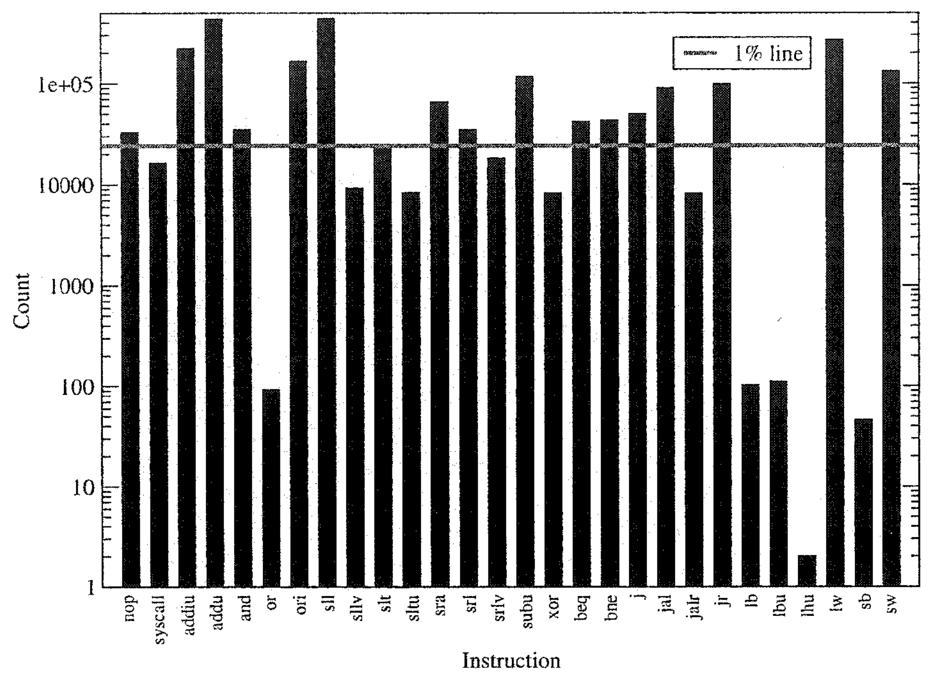

Figure 5.6: Instruction counts for LC trie code with modified ISA

amount. While the IBM instruction set provides only a single branch instruction, ours includes a larger number that allow greater flexibility in conditional and unconditional branching.

There is one instruction that is unique to the IBM instruction set, one of which is the `plo` instruction that determines the position of the leading 1 bit in a register. While it is possible to replicate this operation in software, performing it in hardware is probably much more efficient. It is possible that the designers arrived at this instruction through methods similar to those discussed in Chapter 2.

Most of the instructions in the IBM instruction set have equivalent implementations in the MIPS instruction set. The exceptions are the `plo` instruction, which would require a microcoded implementation in MIPS, and the `out`, `msg`, `lpg`, and `ecp` instructions, which are used for communication with the local processor. The latter four are architecture specific, and do not relate to MIPS.

5.3 Pipeline stalls

Pipeline stalls are related to the structure of both the hardware and the software, and can be reduced by modifying either. A careful consideration of how the software maps to the hardware can provide insight into stalls.

5.3.1 Stalls in the radix lookup

Stalls comprise a large number of the total machine cycles required for routing table lookup on both the original and modified architectures. In fact, over 40% of cycles are stalls in both cases.

Instruction	Stalls	Instruction	Stalls
addiu	319378	beq	6848784
addu	1774301	bgez	3439074
and	5158783	bgtz	71412
andi	344066	blez	16392
nor	50	bltz	124553
or	32812	bne	1344788
sll	32784	jalr	32768
sllv	96	jr	4
slt	325180	lb	2
sltiu	32030	lbu	46298
sltu	186	lh	2539116
srl	64078	lw	106013
srlv	82	sb	18
subu	128	sw	254841
xor	143696		
xori	6		

Table 5.3: Stall cycles for original instruction set

The total number of stalls for the unmodified instruction set is 23 million, which equates to 2814 per lookup. All stalls incurred during execution of the routing code are the result of data dependencies. Although it is possible for the processor to stall for other reasons, such as memory latencies, cache misses, or structural hazards, these problems are never encountered. Memory latency does not have an effect because the simulation assumes zero memory latency. This is unrealistic in practice, however it allows us to model the behaviour of the processor at its data flow limit, and to identify bottlenecks that might otherwise not be apparent. Cache misses do not occur for the same reason. Caches are not used in the simulation because memory already has zero latency, and therefore caches would only have negative performance consequences.

Of the ALU instructions, those that produce the most stalls are addu and and, both of which have very high instruction counts as well. By far the biggest producer of stalls, however, are the branch instructions, with a combined total of 11.8 million, which accounts for over half the total number of stalls. The reason for the high number is quite straightforward, and has to due with the structure of the MIPS pipeline. Consider the following fragment of assembly code.

```
lw  $5,15($6)
beq $5,$0,my_func
```

This sequence of instructions is especially problematic because the result from the first instruction is not available until the MEM stage, whereas the second requires it in ID. Therefore, the ID stage has to stall twice before the result is available to perform the comparison. The assembly code has numerous such cases. The stalls caused by all of the branch instructions are analogous.

The `lh` instruction has a very large number of stalls that can be traced to a similar situation as with the branch instructions. Often, two loads occur in sequence, with the second requiring the result of the first. An example is shown below.

```
lw $5,15($6)
lh $7,10($5)
```

In this example, the address loaded by the first instruction is used as the base address in the second instruction. In software, this corresponds to pointer dereferencing. In some cases the compiler is able to insert an instruction between the successive loads in order to reduce the number of stalls that are necessary. Even in such a case though, one stall is still necessary before the result is available from the MEM stage to be forwarded to the ID stage. In total, the assembly code for the radix algorithms contains 24 instances of dependent `lh` instructions where the two instructions are successive or separated by one independent instruction. The same problems occur with the other load instructions though in fewer cases, and with the sw instruction as well.

The number of stalls for most instructions does not change significantly for the new instruction set, with one exception. Table 5.4 presents the results for the new instruction set.

Instruction	Stalls	Instruction	Stalls
addiu	319418	beq	10359274
addu	1774291	bne	1610354
and	5330816	jalr	32768
or	32832	jr	4
sll	32784	lb	2
sllv	96	lbu	46348
slt	3888811	lh	2539116
sltu	16201	lw	106013
srl	64078	sb	18
srlv	82	sw	254841
subu	128		
xor	143679		

Table 5.4: Stall cycles for modified instruction set

The total number of stalls for the modified instruction set climbs to 26.6 million, or 3247 per lookup, an increase of over 15%. The majority of the increase can be accounted for by a single instruction, namely `slt`. In the original instruction set, this instruction produces 325180 stalls, while in the new one, the value increases to over 3.8 million, almost 11 times greater. The reason for this is straightforward, and has to do with the nature of branch resolution discussed earlier. The four branch instructions that were substituted with macros each use the `slt` instruction to perform a comparison. The result of the comparison is then used by either beq or bne to resolve a branch. The result of `slt` is not available until the end of EX, and branches are resolved in ID, therefore a stall is necessary to wait for the result. This extreme increase in the number of stalls for a single instruction brings into question the viability of the four branch substitutions. If removal of these instructions is very beneficial in terms of hardware complexity, alternate avenues should be explored so that their functionality is not required at all. This may include changes to the algorithms themselves. If the effects of removing the instructions are marginal in terms of hardware, keeping them will be much more efficient if they are used often.

The instructions that are not involved in any new macros do not experience an increase in stalls. This is expected because the routing code has not changed algorithmically from the original instruction set, nor has the compiler. Only those instructions that are used in new macros may experience increases in stall count.

For a 6.7% increase in instruction count from the old instruction set to the new one, there is a 15% increase in stall count, the majority of which come from the `slt` instruction. The percentage of machine cycles that are stalls using the original instruction set is 41%, whereas with the new one it is 44%. The overall increase in cycle count from old instruction set to new is 9%. These figures all respresent the instruction set with the four conditional branch instructions removed.

If the four branch instructions are kept, the changes differ significantly, and the number of stalls is actually less than with the original instruction set. In this case, there is a 0.7% increase in instruction count and a 0.8% decrease in stall count. The net effect on total cycle count is an increase of less than 0.1%! Unless the algorithms can be changed so that they do not require the four branch instructions, it is clearly beneficial to keep them. The reasons for the net drop are the instructions in the original instruction set that take immediate arguments, and the instructions used in the modified instruction set to substitute them. Table 5.5 summarizes this data.

Instruction	**Original**	**Modified**
and	5158783	5330816
andi	344066	-
sltu	186	16201
sltiu	32030	-
slti	0	-
xor	143696	143679
xori	6	-
TOTAL	7453068	7265617

Table 5.5: Stalls for substitutions of instructions with immediate arguments

The reason that the number of stalls is reduced is best explained with an example. Consider the following fragment of assembly code.

```
lw   $2,10($6)
andi $2,$2,0x00ff
```

In this case, the `andi` instruction will experience two stalls while waiting for the result of the `lw`. Recall that result will not be available until the end of MEM, although it is needed by the next instruction in ID. If the `andi` instruction is now substituted with a macro, the following code will be generated.

```
lw  $2,10($6)
ori $1,$0,0x00ff
and $2,$2,$1
```

The insertion of an instruction between the `lw` and `and` eliminates one stall. By the time `and` has reached ID, `lw` is already in MEM, so a wait of only one cycle is required for the result.

5.3.2 Comparison to LC trie

The LC trie algorithm requires an average of 317 cycles per lookup on the original instruction set, only 1/20th that of the radix algorithm. Furthermore, it is much more efficient, in that only 12% of the cycles are stalls, compared to the radix's 41%. With the modified instruction set (without the four branch instructions), the average number of cycles per lookup is 331, a net increase of 4%. The most notable difference is the insignificant change in the stall statistics for the `slt` instruction, although those for `beq` increase significantly. Overall, because of the nature in which the algorithm is implemented, the LC trie approach is far more efficient in terms of stalls. Table 5.6 summarizes the stall data for the LC algorithm on the modified instruction set.

Instruction	Stalls	Instruction	Stalls
addiu	186	beq	83625
addu	139705	bne	53305
and	71	jalr	16384
or	64	jr	16388
sll	142	lb	2
sllv	96	lbu	116
slt	10	lw	49
sltu	201	sb	18
srl	26608	sw	365
srlv	82		
subu	1922		
xor	8207		

Table 5.6: Stalls for LC trie with modified instruction set

5.3.3 Avenues for optimization

The best avenue for optimization is the compiler, because all the stalls discussed above are the result of data dependencies. More sophisticated scheduling is required in order to minimize the number of dependent instructions that directly follow one another. The most benefit would be achieved if the compiler can be made more aware of the pipeline architecture and its behaviour in terms of branching. Given the large number of stalls that are generated due to dependencies on data loaded from memory, improved scheduling could have a significant impact in this area.

5.4 Branches

Jumps and branches account for over 21% of the total number of instructions executed in the radix algorithm. Of the 32 million instructions, 6.7 million are branches. Considering the large number of branches, it makes sense to examine their behaviour to determine whether there is any room for optimization. This section describes issues related to branch performance, and concludes with ideas for how performance might be increased.

Instruction	**Frequency**	**Stalls**
`beq`	3428373	6848784
`bgez`	1719583	3439074
`bgtz`	35707	71412
`blez`	8197	16392
`bltz`	124551	124553
`bne`	736923	1344788
`j`	636594	0
`jal`	24260	0
`jalr`	16388	32768
`jr`	40650	4
TOTAL	6771187	11897775

Table 5.7: Branch instruction frequencies and stalls

Table 5.7 summarizes the data about jumps and branches. The most heavily-used instruction is the `beq` instruction, which accounts for more than half of the total number of branches. The majority of `beq` instances are used for comparison against zero, as are most of the `bne` instances.

Unconditional branches (jumps) account for relatively few of the total number of branches. Procedure calls in the routing code are handled exclusively by the `jal` instruction. The `jalr` and `jr` instructions are not found in the BSD routing code at all, and are only used by library functions.

5.4.1 Branch behaviour in radix and LC trie algorithms

The radix tree routing code contains 238 unique branches in the lookup portion of the code. Of these, 186 are either always taken or never taken. Similarly, of the 180 unique branches in the lookup portion of the LC trie code, 157 are either always taken or never taken. Table 5.8 summarizes these results for the two algorithms.

	Radix	**LC**
Always Taken	88	80
Never Taken	98	77
TOTAL	186	157

Table 5.8: Branch behaviour for radix and LC algorithms

There are two actions that can be taken to optimize the behaviour of such one-way branches. Modifications can be made at an algorithmic level by changing the statements that generate the branches. It is also possible to make changes at the assembly language level by removing the branches that are never taken, and by substituting those that are always taken with jumps. The advantages of the latter may not immediately be obvious. There are two reasons for doing this, however. First, unconditional branches normally do not generate stalls because there are no comparisons that need to be made. Therefore, if the branch instruction is dependent on a preceding load instruction, the stalls can be eliminated. Second, it is possible to implement zero-delay jumps, whereby an instruction decoded as a jump is flushed from the pipeline and replaced with the target instruction in the same cycle. This is discussed in Section 5.4.2.

5.4.2 Opportunities for optimization

While the `jalr` and `jr` instructions exhibit some variation in their target addresses, the `j` and `jal` do not. Furthermore, these two instructions never experience any data dependencies, and because they are unconditional, the branch is always taken. This allows the possibility of implementing zero-delay jumps. Conceptually, if the jump instruction can be resolved in the fetch stage, it can immediately be discarded and the target instruction fetched. This eliminates the cycle used by the jump instruction.

Although branch prediction requires a change to the delayed branching semantics of MIPS, zero-delay jumps do not. The compiler can continue to insert instructions into delay slots. In the event that a jump is resolved in the fetch stage, the subsequent instruction will be fetched to replace it, and then the program counter can be modified to the target address. This technique can also be implemented with a target buffer, except that no prediction bits are required since jumps are always taken. This also eliminates the need for rollback, because mispredictions cannot occur. If a jump instruction is encountered for the first time during the fetch cycle, a hit to the buffer will not occur. The buffer will then be updated in the decode stage as with branch prediction.

5.5 Towards an architecture for IP routing

Routing table lookup is the predominant activity for a router and consumes 80% or more of its CPU time. The primary job of a router is to route packets, and other activities, if handled by the same processor, are secondary. In larger routers, each interface may have a processor for performing lookups, with a central processor that manages updates and other control functions. A large optimization effort is warranted, because packet routing is the most important function. Even a small increase in performance on a single lookup will translate into the possibility for higher throughput, and vendors are under constant pressure to improve throughput. There are two possibilities from a vendor's standpoint. Either they have the ability to change the processor architecture, or they are constrained to a commodity part and must learn to use it most effectively. In the former case, they are free to explore the architectural changes discussed in Section 5.2. In the latter case, an effort to profile the code in detail and understand how it maps to the hardware can lead to improved performance. The results from Sections 5.3 and 5.4 provide insights in this area.

From a hardware perspective, the goal of a small instruction set is to minimize architectural complexity of the processor in terms of chip area, number of functional units, and decoder complexity, all of which may have an effect on the processor's cycle time. The cycle time of a pipelined processor is limited by its slowest stage, so it is conceivable that simplifying one stage can have an effect on others. Cycle time is also limited by propagation delay and other electrical properties. Chip area, therefore, plays a role in performance. Reducing the area of the processor also allows more effective use of existing resources. A smaller ALU would allow more area to be used for caches or register files, for example. Chip area is affected directly when functional units are removed. For example, the floating point unit may constitute a large portion of the logic on a processor. Its removal will free a great deal of area, and the decoder can be simplified by removing the logic that checks for those instructions. When functional units are removed that are not used, the benefits are clear. The software does not have to be modified because it did not require those instructions originally. The effects must be examined more carefully, however, when functional units are removed that are being used. For example, multiplication and division can be emulated in software or microcode, but the architectural advantages should offset any degradation in software performance, otherwise the changes will not be beneficial. This was observed clearly with the removal of four branch instructions from the R2000 ISA, which caused the stall count of the `slt` instruction to increase by a factor of eleven (Section 5.3.1). In this case, it is more efficient to keep those instructions if the compiler or algorithms cannot be changed to not rely on them.

In general, a smaller instruction set will result in increased code size as more functionality has to be implemented in software. The BSD code experienced a 2.5% increase in code size with the modified instruction set. As a result, more instruction words need to be fetched from memory, which can cause a problem if memory access latencies are high. By incorporating a larger instruction cache onto the chip, however, this problem can be dealt with, a feat made possible by reducing the complexity of the processor. A possible advantage of a smaller and simpler instruction set is better compiler optimization. With the larger code size resulting from a simpler instruction set, the compiler may have more freedom in manipulating and reordering instructions in order to minimize stalls and best utilize existing resources.

Based on our results, the only functional units that are required for both the radix tree and LC trie algorithms are an integer adder, a shifter, and a load/store unit. Floating point operations are not required, which eliminates the need for such a unit. Integer division is not required by the routing algorithms themselves, but only in very limited amounts by some library functions. Modification of the algorithms or the functions to emulate division could remove the need for division instructions. Similarly, the only instances of integer multiplication occured in the driver program, so in reality these instructions are not required either.

With any processor that is used for a specific class of algorithms, an effort should be made to ensure the algorithms make effective use of the hardware and that the architectural mapping is efficient. The key is to ensure that enough of the right functional units exist to minimize structural hazards, and that the pipeline is organized in such a way that data hazards are minimized. In the case of the R2000, the routing algorithms make effective use of the processor resources, although it is possible that some structural changes and additions would be beneficial. These changes are suggested by a number of performance concerns. Some of these can be mitigated with improved compiler scheduling. For example, the BSD code contains many instances of memory-to-memory move operations where a value is loaded from memory and immediately stored to a different location. In most cases, the compiler makes no attempt to schedule instructions between the load and the store to eliminate the data hazard. If such scheduling is not possible, it would be conceivable to add a write port in the writeback stage and to forward the value to that stage once it is available.

Performance bottlenecks also arise from load-branch stalls as discussed in Section 5.3. Once again, improved compiler scheduling can be used to manage this, otherwise the pipeline can be restructured. Figure 5.7 shows a modified pipeline whose MEM stage operates in parallel with EX. This has implications for all instructions, not just loads and stores. Other instructions pass through the memory stage unchanged, which is a wasted cycle. If the CPI is 1, this is not an issue. In fact, CPI is greater than 1, however, which is the result of data dependencies such as the load-branch type. If effective address calculation can be moved to the ID stage, then one stall can be eliminated because the branch instruction will only need to wait one cycle for the data value, not two, reducing the cycle count of the radix code and LC code by 723 cycles and 6 cycles per lookup, respectively. Moving effective address calculation to the decode stage requires an additional adder, however, which increases the complexity of the processor.

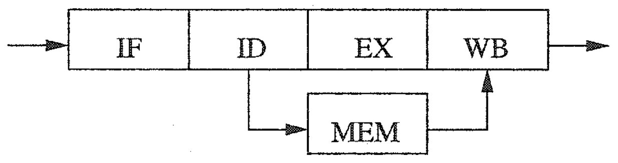

Figure 5.7: Restructured R2000 pipeline

Another possible structural change involves the addition of a specialized increment unit. A large portion of the addition that takes place in the BSD code involves incrementing by one. The general-purpose adder represents a significant bottleneck along the critical hardware path, and may be the major reason that cycle time cannot be reduced. If the adder is taken off the critical path and transformed into a multi-cycle functional unit by means of a separate pipeline, it is conceivable that an increment unit could run at a higher speed. An example of such a pipeline is shown in Figure 5.8. While compiler scheduling can be used to solve data dependency issues, it cannot be used to decrease cycle time, which makes this approach of using a more complex pipeline attractive. As long as the

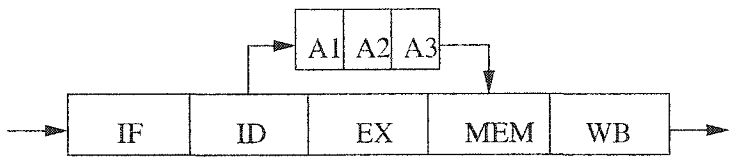

Figure 5.8: Pipeline with multi-cycle adder

gains achieved by the increment unit offset the losses of a multi-cycle adder, this method would be effective. A more detailed gate-level analysis is required to determine the exact benefits and trade-offs of such an implementation.

The remaining issue that needs to be examined is whether there are any new instructions that would be useful in the instruction set. The bulk of the related work has concentrated on generating instruction set extensions for specific applications by looking for frequently recurring groups of operations. If any such instruction set extensions have performance advantages, they should be considered.

Considering the nature of routing table lookup, an architecture with very fast memory access and large caches is very important. Freeing up as much area as possible on the processor allows more space to be devoted to data caches, which allows much faster data accesses, because cache hits will be more frequent. As cycle times decrease, signal propagation delay becomes increasingly important, which emphasizes the need to pack the processor resources into as tight a space as possible. Our experiments assumed zero memory latency, and as such the results are those of a processor operating at its data flow limit. In reality, memory access times are nonzero, and cache misses occur. Thus, it is important to maximize their performance to keep CPI as low as possible. Larger register files are also important to compiler optimization. Rather than continually accessing memory, important variables can be kept in registers, which can cut runtime by up to one half [1].

From the standpoint of software, there are a number of avenues that a vendor can explore to optimize the efficiency of their product. The performance of the routing codes is heavily affected by stalls. Over 40% of all machine cycles in the BSD code are wasted because of stalls. For the LC trie algorithm, 12% of cycles are stalls. In both cases, the instructions that are the largest producers of stalls are branch instructions, which account for more than 50% of the total number in the BSD code, and almost 30% in the LC trie code. These stalls arise out of the load-branch dependencies discussed earlier, and can be reduced or eliminated with better compiler scheduling. Given that branches are resolved two stages prior to data values being returned from memory, they are particularly susceptible to delays and this has to be taken into careful consideration when designing a compiler. As pipelines grow deeper, this problem can become more serious. Another area where this is problematic is with pointer dereferencing. Such operations require two consecutive loads, where the second is dependent on the result of the first. If the compiler does not schedule any operation between the two loads, the second will stall for two cycles in the decode stage waiting for its arguments. Not surprisingly, this is more prevalent in the BSD code, where almost 12% of stalls are of this nature. The LC trie algorithm, which requires much less pointer manipulation, experiences this problem a negligible number of times.

In general, a combination of improved scheduling and code analysis can have an affect on all of the stalls discussed in Section 5.3, because they are all the result of data dependencies. The BSD code stands to receive up to a 40% performance gain should stalls be eliminated or reduced to negligible levels.

Branches are another software avenue that can be explored in order to attain higher efficiency. In the BSD code, 78% of all unique conditional branches are either *always* taken or *never* taken. These branch instructions account for 11% of the total number of branches, or about 2% of the total number of cycles. The same can be said for 87% of the branches in the LC trie code, which account for 49% of its total branch count, and again about 2% of its total cycle count. Branches of this nature can represent tests in the code for conditions that never occur, such as error checking. Once software has been verified, removing such statements can increase performance. Not only will the cycle corresponding to the branch be eliminated, but if the branch depends on the result of a previous load instruction, the corresponding stall cycles can be eliminated as well. A zero-delay jump scheme can also be used to further boost performance. The `j` and `jal` instructions account for 2% of the total instruction count in the radix code, and 6% in the LC trie code. Theoretically, if all of these instructions can be replaced with the jump targets without a delay, 80 cycles per lookup can be saved from the radix algorithm, and 17 from the LC algorithm.

Considering both software factors, the cycle count of the BSD code can be reduced by up to 43% on the original architecture. That translates into 3846 cycles per lookup compared to the original 6747. For the LC trie algorithm, cycles per lookup could be reduced to 255 from the original 317. This is still close to double the commonly accepted value of 150 instructions per lookup. An effective optimization strategy involves efforts at the algorithmic level and during compile-time and code generation. To put these changes into perspective, a 3 GHz processor running the radix code could process 780000 packets per second, up from 444600. The same processor running the LC trie code could conceivably process 11.7 million packets per second, up from the original 9.4 million, an increase of 24%. In terms of throughput, with 64 byte packets, this translates into an increase of nearly 1.2 Gbps for a total of 6 Gbps. The theoretical maximum throughput is dependent only on the cycle time of the processor and latency of memory, assuming cycles per lookup can be lowered to 1. In this case, the same hardware could process 3 billion packets per second for a total throughput of 1.5 Tbps, given zero memory latency.

5.6 Concluding remarks

Optimizing an architecture for routing amounts to architectural optimization that minimizes cycle time, and hardware and software optimization that lowers CPI to as close as possible to 1. Hardware performance is dependent on the instruction set, the pipeline architecture, the chip area of the processor, and the length of the critical path through the processor. Any one bottleneck in a pipeline stage affects the performance of the entire pipeline, and the goal is to streamline the datapath to eliminate as many such bottlenecks as possible. Restructuring the pipeline can be valuable in customizing the architecture to the needs of the application. In the case of routing, which is heavily dependent on memory access, data reads must be optimized to reduce delays. Reducing the size of the instruction set eliminates unneeded functional units, freeing up chip area, giving more resources to caches or register files. Circuit complexity can also be reduced, possibly allowing reduced cycle times.

From a software standpoint, the compiler needs to be designed to make the most efficient use of the hardware. This requires a knowledge of the innate bottlenecks of the hardware, and how they can be countered by optimal scheduling. Effective scheduling reduces the need for structural changes to the pipeline. A large portion of the stalls that arise in the routing algorithms can be eliminated by more effective scheduling. Branching is also a performance concern that can be addressed. A zero-delay jump scheme can be implemented to eliminate many of the delays from unconditional branches. Additionally, the majority of conditional branches are either always taken or always not-taken, which means the software can be modified to remove them and save cycles. Given the ability of the compiler to schedule instructions into all the branch delay slots, branch prediction is not useful in the R2000 architecture.

Whether hardware vendors have the ability to change architectural features of the processor, or are limited to software changes, there are a number of performance-limiting factors that can be explored in both areas, resulting in performance gains of up to 24% to 43%, depending on the routing algorithm.

Chapter 6

Conclusion

6.1 Summary

A variety of methods have been proposed to generate application-specific instruction set extensions for existing architectures. All of these techniques increase the instruction set size without first examining the existing instruction set to determine its usefulness. Our work differs in that it attempts to determine an instruction set of minimal size that can be used to run IP routing algorithms. What we discovered is that even simple RISC processors provide needlessly complex operations for routing.

From a hardware standpoint, removal of unneeded instructions simplifies the logic by eliminating unused functional units. This has direct effects on VLSI chip area, reducing the area required for the processor itself, and making more available for caches or register files, which has significant advantages for algorithms such as routing that require frequent memory access.

From a software standpoint, the routing algorithms need to be examined carefully to see how they map to the hardware, and the compiler needs to be aware of the hardware's limitations and bottlenecks. Better compiler design can reduce the need for hardware reconfiguration. Additionally, the algorithms themselves can be simplified to increase their performance on the existing hardware. Our experimental data highlights several such areas.

The scope of our work is an architectural-level analysis and we do not examine performance issues at a more detailed gate-level. FAST is unable to estimate processor cycle times and circuit complexity. FAST models behaviour at an artifact level, and changes to the pipeline or instruction set architecture have no effect on its cycle time. It is therefore difficult to determine exactly what effect the removal of an instruction will have in terms of circuit complexity, chip area, and overall execution time. Performing a more detailed gate-level analysis is the next logical step.

6.2 Contributions

This thesis has contributed in three areas. The first and second concern architectural optimization for IP routing, while the third examines the software aspect of the routing algorithms and their deficiencies.

6.2.1 An optimized ISA for routing

We have proposed a minimal ISA for IP routing that was experimentally verified with two different routing algorithms. The instruction set is based on the MIPS ISA and and is 50% smaller than the original. It contains 14 arithmetic/logic instructions, six jump and branch instructions, eight load/store instructions, and one special instruction. Table 6.1 summarizes the instruction set. The instructions that are removed are substituted with assembler pseudo instructions that typically expand to one two (and at most four) machine instructions during assembly. Thus, the functionality of these instructions has been preserved.

Arithmetic/Logic Instructions	**Branch Instructions**
`addu`	`beq`
`addiu`	`bne`
`and`	`j`
`or`	`jr`
`ori`	`jal`
`sll`	`jalr`
`sllv`	
`slt`	**Load/Store Instructions**
`sltu`	`lb`
`sra`	`lbu`
`srl`	`lh`
`srlv`	`lhu`
`subu`	`lw`
`xor`	`sb`
	`sh`
Special Instructions	`sw`
`syscall`	

Table 6.1: Instruction set for IP routing

Both the BSD radix tree algorithm and the LC trie algorithm map effectively to the arithmetic and logic instructions. In its existing form, the radix tree algorithm operates most efficiently if four additional branch instructions are kept: `bgez`, `blez`, `bgtz`, and `bltz`. The LC trie algorithm, on the other hand, almost never uses these and they can be removed without compromising efficiency. Similarly, for the load/store instructions, `lh` and `sh` are not required by the LC trie algorithm.

With the instruction set in Table 6.1, the radix tree code experiences a 7% increase in cycle count. If the four branch instructions are kept, the lookup process experiences a negligible 0.1% increase in cycle count. Since these four instructions are largely irrelevant to the LC trie algorithm, results with or without them do not differ. With the modified instruction set, the lookup process for the LC trie algorithm experiences a 4% increase in cycle count, although the total number of cycles is only 1/20th that of the radix tree algorithm.

The major implications of a reduced instruction set are the lack of requirements for multiply/divide unit and a floating point unit. Removal of both of these can translate into a significant portion of the chip area. This leaves more room for register files and caches, both of which are important for algorithms that frequently access memory. Larger caches will translate into fewer cache misses, and a larger register file can allow the compiler to keep frequently-used variables in registers rather than having to move them to and from memory.

6.2.2 Possible architectural changes

Should improved compiler scheduling prove to be difficult, we have identified several architectural changes that could increase the efficiency of the existing code. The aforementioned stalls due to branching occur due to preceding load instructions. One of these two stalls could be eliminated by converting the MEM stage into a separate pipeline that runs in parallel with the EX stage. Another architectural feature that may be a bottleneck is the general-purpose adder. If this function can be removed from the EX stage and converted into a separate multi-cycle functional unit, it is conceivable that the remainder of the EX stage could operate at a higher clock rate, thus increasing speed for other instructions. These changes are discussed in more detail in Section 5.5.

6.2.3 Software issues

We have identified various software issues in the code that cause performance degradation. Some of these can be removed by changing the algorithms themselves, while others can be solved through improved compiler optimization. The major software problem is the large number of machine cycles that are wasted in the form of stalls. In our simulations, these all arise out of data dependencies. Over 40% of cycles for the radix tree lookup are stalls, and 12% of the cycles for the LC trie lookup are stalls. Section 5.3 breaks these into the number of stalls for each instruction, but in both cases branch instructions are major culprits, accounting for 50% and 30% of the total number of stalls in the radix and LC trie code, respectively. In general, these stalls can be eliminated through improved scheduling, so that results of load instructions are available by the time dependent subsequent instructions enter the decode stage.

Branches incur a cost during the execution of code, and designers need to pay special attention to when and where they are used. We have found that in the BSD code, 78% of conditional branches are either always taken or always not taken. Likewise, for the LC trie code, the value is 87%. Those branches that are always taken can be replaced with jumps to eliminate the comparison. Those that are never taken can be removed completely and the branch cycle and any potential stalls can be eliminated completely. If combined with the zero-delay jump scheme discussed in Section 5.4.2, optimization of branches can reduce the cycle count of the radix code by a further 3%, and that of the LC trie code by over 7%.

6.3 Directions for future work

The work presented in this thesis is a first step toward an architecture for IP routing. A number of additional avenues can be explored to develop a more efficient architectural framework for this application. These areas include:

- **Realistic memory latency and workloads.** Our simulations currently assume zero memory latency in order to model behaviour at the data flow limit. In real life, memory latency plays an important role and needs to be considered. In addition, while our simulations used actual routing tables from Internet core routers, IP addresses were generated at random. The use of workloads captured at the same routers would add an additional level of realism and accuracy to the simulations.

- **Better compiler scheduling.** Many of the architectural pressures can be relaxed if the compiler can schedule instructions around them. With a more complex instruction set this may be more difficult due to smaller code size. A smaller instruction set results in a somewhat larger code size, however, giving the compiler more freedom in reordering instructions to work around architectural challenges. A significant optimization effort is warranted since the routing software consumes the dominant portion of the machine time. A small increase in performance for a single lookup can translate into much larger overall throughput.

- **Creating application-specific instructions.** Now that a minimal instruction set is being used, it may be profitable to apply some of the techniques from Chapter 2 to generate application-specific instructions. Searching for frequently recurring groups of operations may provide insight into functional units that might be useful for increasing performance.

- **Developing a gate-level architectural model.** FAST allows us to experimentally observe the effects of architectural changes on the terms CPI and I from Equation 5.1. It does not provide any insight into the third term, T, however, and as a result, we do not have the complete picture of what effects our changes have. A detailed gate-level model in a hardware description language would allow more concise simulation to determine if cycle time actually decreases due to our architectural changes. This, combined with the previous item, will provide a much more thorough framework for a routing microarchitecture.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Bell Telephone Laboratories, Inc, 1986.

[2] H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Dept. of Information Systems, Kyushu University, January 1996.

[3] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 526 – 532, 1993.

[4] The MIPS Technologies Inc. processor core roadmap. AMSLink: `http://www.amslink.com/pdf/RdmpBack.pdf`, October 1998.

[5] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction set extensions under microarchitectural constraints. In *Proceedings of the 2003 Design Automation Conference*, pages 256 – 261, June 2003.

[6] P. Athanas and H. Silverman. Processor reconfiguration through instruction set metamorphosis: Compiler and architecture. *IEEE Computer*, 13(3):11 – 18, March 1993.

[7] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA language - version 4.1. Technical report, Technical Report, Computer Science Dept., University of Dortmund, September 1994.

[8] J. P. Bennet. *A methodology for automatic design of computer instruction sets*. PhD thesis, University of Cambridge Computer Laboratory, 1988.

[9] P. Biswas and N. Dutt. Greedy and heuristic-based algorithms for synthesis of complex instructions in heterogeneous-connectivity-based DSPs. Technical Report 03-16, Department of Information and Computer Science, University of Californa, Irvine, May 2003.

[10] P. Bose. *Instruction set design for support of High-Level languages*. PhD thesis, University of Illinois at Urbana-Champaign, 1983.

[11] H. Choi, I. Park, S. Hwang, and C. Kyung. Synthesis of application specific instructions for embedded DSP software. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*, pages 665 – 671, 1998.

[12] M. Freericks. The nML machine description formalism. Technical Report 1991/15, Fachbereich Informatik, TU Berlin, 1991.

[13] J. Gyllenhall. A machine description language for compilation. Master's thesis, Dept. of EE, UIUC, IL, 1994.

[14] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of 34th Design Automation Conference*, pages 299–302, 1997.

[15] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of 36th Design Automation Conference*, pages 927–932, 1999.

[16] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design, Automation and Test in Europe*, pages 485 – 490, 1999.

[17] F. Haney. *Using a computer to design computer instruction sets.* PhD thesis, Carnegie-Mellon University, 1968.

[18] M.R. Hartoog, J.A. Rowson, P.D. Reddy, S. Desai, D.D. Dunlop, E.A. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of 34th Design Automation Conference*, 1997.

[19] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA 94101-3205, 1996.

[20] B. Holmer. *Automatic design of computer instruction sets.* PhD thesis, University of California, Berkeley, 1993.

[21] I. Huang and A. Despain. Synthesis of instruction sets for pipelined microprocessors. In *Proceedings of 31st ACM/IEEE Design Automation Conference*, pages 5 – 11, 1994.

[22] M. Imai, A. Alomary, J. Sato, and N. Hikichi. An integer programming approach to instruction implementation method selection problem. In *Proceedings of the conference on European design automation*, pages 106 – 111, 1992.

[23] Internet performance analysis measurement project. `http://www.merit.edu/~ipma`, March 2004.

[24] Gerry Kane. *MIPS RISC Architecture.* Prentice-Hall, Inc, Englewood Cliffs, NJ 07632, 1989.

[25] J. Lee, K. Choi, and N. Dutt. Automatic instruction set design through efficient instruction encoding for application-specific processors. Technical Report 02-23, Center for Embedded Computer Systems, University of California, Irvine, May 2002.

[26] R. Leupers. *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[27] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[28] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa. Advanced processor design using hardware description language AIDL. In *Proceedings of ASPDAC*, 1997.

[29] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[30] Soner Onder. Architecture description language user's manual. Technical report, Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931-1295, 2001.

[31] Soner Onder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of IEEE International Conference on Computer Languages (ICCL98)*, pages 80 – 91, May 1998.

[32] J. Park, S. Vassiliadis, and J.G. Delgado-Frias. Flexible oblivious router architecture. *IBM Journal of Research and Development*, 39(3):315 – 329, May 1995.

[33] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA — machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of 36th Design Automation Conference*, pages 933–938, 1999.

[34] Johan Van Praet, Gert Goossens, Dirk Lanneer, and Hugo De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of 7th IEEE/ACM International Symposium on High-Level Synthesis*, pages 11 – 16, May 1994.

[35] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of 11th International Symposium on Systems Synthesis*, pages 31 – 36, 1998.

[36] K. Sklower. A tree-based packet routing table for berkeley UNIX. In *Proceedings of USENIX*, pages 93–103, 1991.

[37] R. Sweet and J. Sandman. Emperical analysis of the mesa instruction set. In *Proceedings of ASPLOS*, pages 158 – 166, 1982.

[38] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system-on-chip design. In *Proceedings of 6th Asia Pacific Conference on Chip Design Languages*, pages 109 – 116, 1999.

[39] The MDES user manual. Trimaran Release: `http://www.trimaran.org`, 1998.

[40] John Waldron. *Introduction to RISC Assembly Language Programming*. Addison-Wesley Longman Ltd, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 1999.

[41] x86. `http://en.wikipedia.org/wiki/X86`, March 2004.

Appendix A

Implementing branch prediction

In most pipelined processors, instructions are decoded and operands are fetched prior to the branches being resolved. In many cases, an ALU operation is necessary to resolve the branch, which usually takes place in a later pipeline stage. This necessitates a pipeline stall, since instruction fetch cannot proceed until the branch has been resolved and the address of the next instruction is known. Unfortunately, the result of this is a waste of machine cycles. The cycles between branch decoding and branch resolution go unused, and any pipeline stages between those where these activities take place are filled with bubbles.

In order to counteract this problem, it is necessary to add support for dynamic hardware prediction. The simplest form of dynamic prediction is a branch history table that is indexed by the lower order bits of a branch address. Each location in the table corresponds to one bit that keeps track of whether the branch was last taken or not. The obvious shortcoming of this is that the prediction may actually correspond to a different branch. From the viewpoint of the hardware, though, there is no difference. If the prediction turns out to be incorrect, the bit is inverted. Another shortcoming of using only one bit is that prediction accuracy may suffer when a branch heavily favours taken or not taken, such as in loops. For example, if a branch that strongly favours taken is not taken only once, there will be two mispredictions, since the predictor will first predict taken, then will be updated, and will then predict not taken. The solution to this problem is to add more bits to the predictor. The general case of this is an n-bit saturating counter, whereby the it predicts not taken for values between 0 and $2^n - 1$, and predicts taken otherwise. While a 2-bit predictor requires little additional hardware over a 1-bit predictor, studies have shown that they perform almost as well as predictors with more bits [19]. Figure A.1 shows the state diagram for a 2-bit predictor. Notice the branch must be taken twice before it predicts taken, and once the counter is saturated, there must be two mispredictions before it is predicted not taken.

This system by itself works effectively in pipelines where instructions are decoded before branches are resolved. Unfortunately, in the R2000 pipeline, branch resolution takes place in the decode stage. If prediction is to have any use, it must take place in the fetch stage before instructions are decoded. The problem with this is that the target address remains uncomputed until the decode stage, therefore

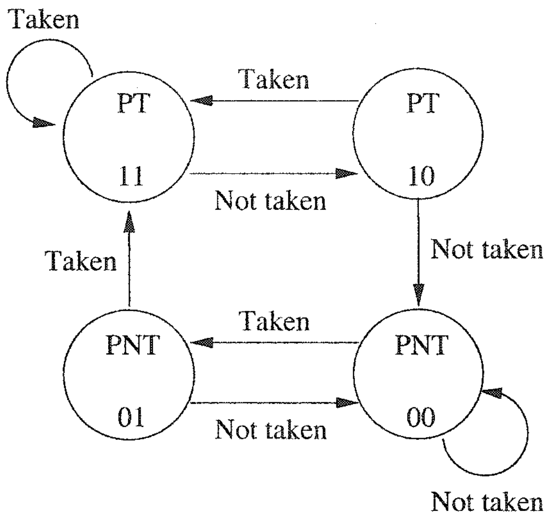

Figure A.1: 2-bit Branch predictor

a means is required for storing these addresses. A branch *target buffer*, accessed during the fetch stage, is used to store target addresses of instructions. Figure A.2 depicts a target buffer.

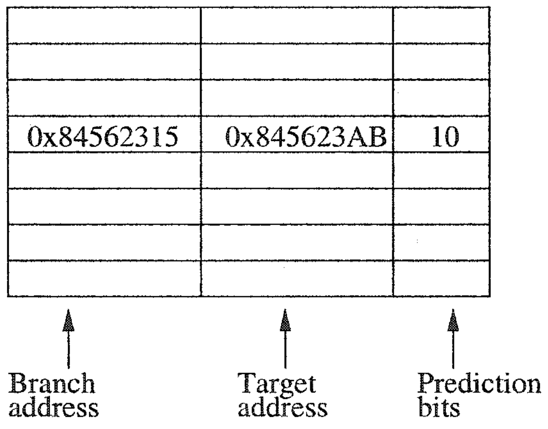

Figure A.2: Branch target buffer

Each entry in the target buffer contains the address of the branch, the predicted address, and the prediction bits. Each instruction that is fetched is checked against the target buffer. If the low order bits of the address index to a location that contains the address of the branch, a prediction is made based on the value of the prediction bits. If the branch is predicted taken, fetching immediately continues from the target address in that location of the buffer. The addresses of the branches must be stored since prediction takes place prior to instruction decoding. If this check was not made, it is possible that predictions would be made for instructions that are not branches, which would have a negative impact on performance.

A combination of branch target buffer and 2-bit predictor was chosen for branch prediction implementation. Changes are required to both the instruction fetch (IF) and instruction decode (ID) stages. These changes, along with how the target buffer and predictor are implemented, are discussed below.

A.1 Implementation of predictor and target buffer

The 2-bit branch predictor is implemented according to the simple finite state model depicted in Figure A.1. The code for the predictor is given in Section 4.3.1. It is implemented as a custom ADL artifact that has a user-defined number of entries. The artifact declares an array of `size` entries, each 2 bits wide. When called as an r-value, the artifact simply returns the value in the entry to which the address argument indexes. When called as an l-value, the artifact sets the value of the bits at the respective index. When given an argument of 1, the predictor bits are incremented, and when given an argument of zero, the bits are decremented.

The implementation for the target buffer is shown in Figure A.3. Statistics collection is omitted for the sake of brevity. The artifact maintains two arrays, one for the branch instruction address and a second for the target address. When called as an r-value, the index is calculated and the target address returned if the instruction address stored in the buffer corresponds to the address used to index into the buffer. Otherwise a value of zero is returned to signal a mismatch. The buffer is updated with the provided arguments when called as an l-value.

While it would have been possible to implement the target buffer and predictors as one artifact, two separate artifacts were chosen for more flexibility.

A.2 Changes to Instruction Fetch stage

The IF stage is responsible for both making predictions and handling mispredictions. Predictions involve checking the target buffer and examining the prediction bits. The code for this is shown below, and is located in the IF prologue.

```
if (tbuf[pc] == 0) then
   begin
      predict = 0;
   end

   else                      # Prediction occuring
   begin
      # Check the 2-bit predictor
      if (bp[pc].[1:1] == 0) then
         begin
            predict = 0;     # Predict not taken
         end
         else
         begin
            predict = 1;     # Predict taken
```

```
artifact target_buffer attributes (size)
begin

   # The necessary storage
   integer array instruction_address[size, 32];
   integer array branch_address[size, 32];
   integer hash_val;

   # Initialize the artifact when created
   initialization
   begin
      forall instruction_address=0;
      forall branch_address=0;
   end initialization;

   # rvalue procedure: called when artifact is an rvalue
   rvalue(addr)
   begin

      # Calculate the hash value
      hash_val = addr.[10:8];
      rvalue = 0;
      if (instruction_address[hash_val] == addr) then
         rvalue=branch_address[hash_val];

   end rvalue;

   # lvalue procedure: called when artifact is an lvalue
   lvalue(addr, br_addr)
   begin

      # Calculate the hash value
      hash_val = addr.[10:8];

      instruction_address[hash_val] = addr;
      branch_address[hash_val] = br_addr;

   end lvalue;

end target_buffer;
```

Figure A.3: Branch target buffer artifact

```
            predict_target = tbuf[my_pc];
         end;
   end;
```

The first condition determines whether the target buffer returns a value of zero for the current value of the program counter. This can occur under two circumstances, either before it has been initialized with its first branch address, or if the current instruction does not correspond with a branch instruction. If neither of these are the case, that means the current instruction is a branch instruction. The value of the corresponding prediction bits is then examined. Only the high order bit needs to be checked, since values of 00 and 01 correspond to predict not taken, and 10 and 11 correspond to predict taken. If the branch is predicted taken, the target is set to the value contained in the buffer.

In the event of a misprediction, the IF stage is responsible for handling *rollback*, which amounts to flushing the IF stage to nullify the incorrectly fetched instruction. This is handled in the epilogue minor cycle, the code for which is shown below.

```
if (rollback) then
begin
   rollback = 0;
   retire nostat;
   pc = miss_pc;
```

```
      miss_pc = 0;
      newcontext;
   end;
```

The rollback flag is set in the instruction decode stage, and will be discussed later. In the event of this condition, the flag is reset, and the instruction context is retired without generating statistics. This has the effect of generating a pipeline bubble. The program counter is then set to the address of the instruction following the branch, stored in miss_pc in ID. A new instruction context is then generated to fetch the next instruction from memory.

If rollback did not occur in this cycle, and a prediction was made, the next instruction has to be fetched from the predicted address. This occurs in the epilogue as well, and takes place with the following code.

```
if (predict) then
begin
   branch_input = 0;
   pc=predict_target;
end;
```

The branch_input flag is reset until it is required by the next branch instruction.

A.3 Changes to Instruction Decode stage

Whereas the IF stage is responsible for checking the target buffer and handling rollback, the ID stage is where the buffer is updated and where the need for rollback is detected. All activity takes place in the intermission minor cycle. The first piece of code handles the case where the instruction is a branch and the there was a prediction of not taken[1].

```
if (predict == 0) then
begin
   if (c_what ^= condition_z) then
      branch_target=my_pc + sign_extend_14(immediate);

   do_forwarding_to_id;
   condition_code(lop,rop);

   ### NO PREDICTION -- BRANCH TAKEN ###
   if (branch_input) then
   begin
      tbuf[my_pc] = branch_target;
      bp[my_pc] = 1;
      rollback = 1;
      miss_pc = branch_target;
      branch_input = 0;
   end

   ### NO PREDICTION -- BRANCH NOT TAKEN ###
   else
   begin
      bp[my_pc] = 0;
   end;
end
```

[1] In this implementation, predict not taken is equivalent to no prediction at all. The following code handles both cases.

If no prediction was made but the instruction is a branch, then the target address is calculated from the sign-extended immediate portion of the instruction. The condition_code procedure performs the comparisons to determine whether the branch is taken or not and sets the branch_input flag. If the branch is to be taken, this corresponds to a misprediction, so the target buffer is updated with the new branch target, the predictor is incremented, and the rollback flag is set. If the branch is not to be taken, then the predictor bits are decremented.

If a prediction was made, then a check must be made as to whether the prediction is correct or not. This is handled by the following code.

```
else
begin
   if (c_what ^= condition_z) then
      branch_target=my_pc + sign_extend_14(immediate);

   do_forwarding_to_id;
   condition_code(lop,rop);

   ### CORRECT PREDICTION -- BRANCH TAKEN ###
   if (branch_input) then
   begin
      # Check if the two targets are equal - if not, rollback
      if (predict_target ^= branch_target) then
      begin
         rollback = 1;
         bp[my_pc] = 0; bp[my_pc] = 0;
         tbuf[my_pc] = 0;
         miss_pc = branch_target;
         branch_input = 0;
      end

      else
      begin
         tbuf[my_pc] = branch_target;
         bp[my_pc] = 1;
         branch_input = 0;
      end;
   end

   ### INCORRECT PREDICTION -- BRANCH NOT TAKEN ###
   else
   begin
      bp[my_pc] = 0;
      rollback = 1;
      miss_pc = my_pc + 4;
   end;
end;
```

If a prediction was made correctly, a check takes place to see whether the predicted target is equal to the actual target. This is required due to the jr and jalr instructions, whose target addresses are contained in registers and can vary during program execution. If the targets are not equal, rollback occurs and the branch instruction is removed from the target buffer. Otherwise, the the predictor is incremented. If a misprediction occurred, then the predictor is decremented and the rollback flag is set. The value of miss_pc is set to be the address of the instruction following the branch instruction. The fetch stage then discards the incorrectly fetched instruction and restarts from the

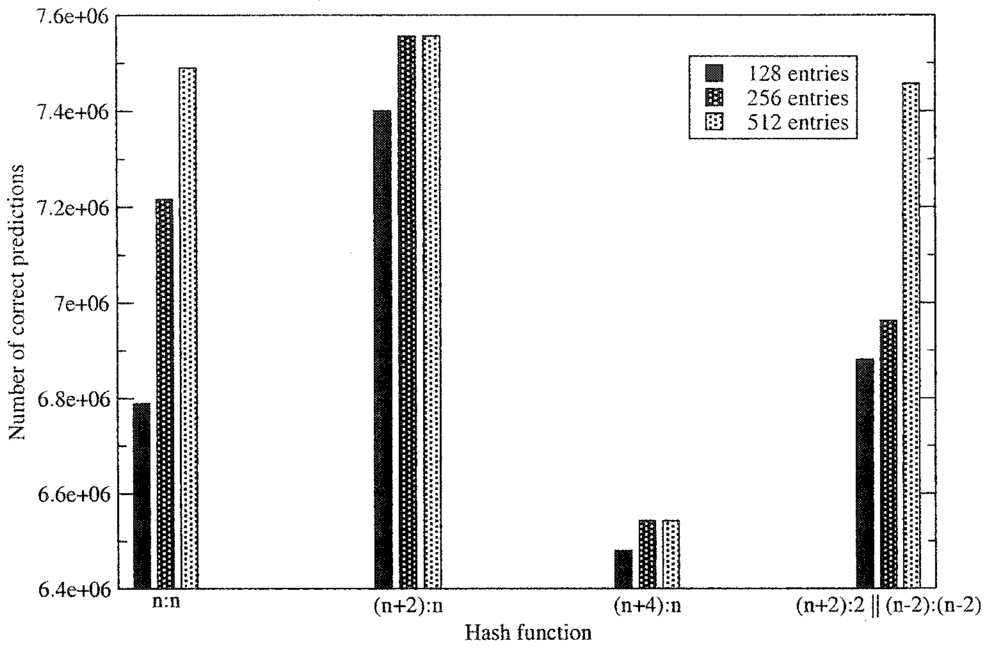

Figure A.4: Number of correct predictions varying table size and hash function

correct address.

A.4 Results of branch prediction

Adding branch prediction to the R2000 requires a fundamental change to its delayed branching semantics. Recall from Chapter 3 how the compiler attempts to insert instructions into the delay slots following branch instructions, since the instruction immediately following a branch is fetched prior to the branch being resolved. The compiler attempts to place an instruction in this location that should be executed whether the branch is taken or not. This essentially amounts to a static strategy of always predicting not taken, although there are no penalties if the prediction is incorrect. Since the instruction in the delay slot must always be executed, there is no need for rollback or pipeline flushes. Adding dynamic hardware prediction requires this be changed, since a benefit will only arise if the delay slot is eliminated, or if branches are resolved in a later pipeline stage. Since the pipeline architecture is not changed, the delay slot must be eliminated for any changes to be observed.

A simple way of verifying the behaviour of the branch prediction is to compile the routing code without optimizations. Instead of inserting useful instructions into the delay slots, the compiler simply inserts nops, which can safely be ignored. The following results are the totals from both routing table construction and lookup.

Figure A.4 shows the number of correct predictions that occur subject to a variety of buffer sizes and hash functions. The value of n is equal to log(buffer_size), and the format of the hash functions are standard ADL syntax. For example, $(n+4):n$ for a buffer size of 256 represents 8 bits starting

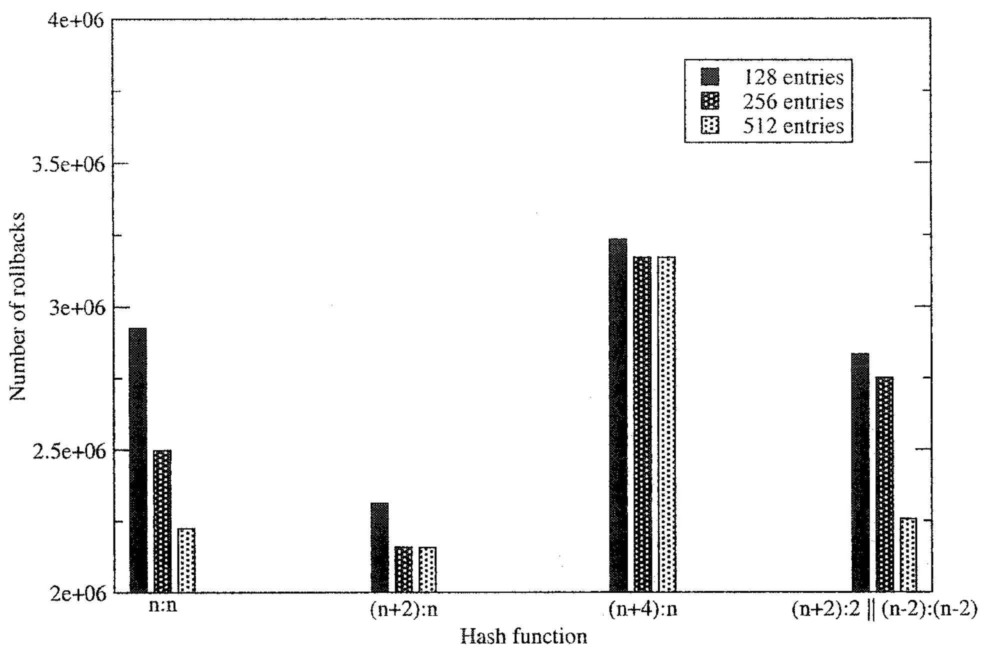

Figure A.5: Number of rollbacks varying table size and hash function

at bit 12 (bits 4 through 12). Figure A.5 displays the number of rollbacks (or mispredictions) that occur for the same choices of buffer sizes and hash functions. As should be expected, the two graphs are essentially inverses of one another.

Both graphs show that an increased table size is an advantage. A larger table allows more state to be maintained about branches. This translates into less collisions and greater prediction accuracy. The difference between the buffer sizes is most pronounced with the first function that uses the low order bits. The differences are least pronounced with the third function that uses higher-order bits. The performance of the third function is much poorer, however. Ignoring the lowest order four bits results in less variation among addresses in localized areas of the code. Consequently, there is likely a great deal more contention for buffer locations resulting in poorer accuracy.

The second function provides the best performance for all three buffer sizes. Eliminating the two lowest order bits appears to be ideal. Instructions are aligned on word boundaries, which are every four bytes. Therefore, the level of precision provided by the first two bits is not required. Furthermore, a processor is most likely to benefit from a branch prediction scheme that can differentiate between addresses in a localized region of code. The inclusion of higher-order bits, as in function four, is therefore not as useful.

The highest number of correct predictions is roughly 7.5 million when using the second function with a 512 entry buffer. The same function and buffer size results in 2.2 million rollbacks. The total number of cycles required for execution of the unoptimized routing algorithms is 76 million. Each correct prediction saves one cycle, while each rollback costs a cycle, bringing the net savings to 5.3

million cycles. This is roughly a 7% performance improvement.

Branch prediction will only be useful if it provides a benefit on optimized code. For this to happen, the compiler has to be modified so that it does not create a delay slot. Furthermore, branch prediction in the R2000 architecture would only be useful if the compiler has difficulty filling the delay slot in some instances. If the delay slot can always be filled, then branch prediction in the R2000 is not useful. Changing the delayed branching semantics will ultimately be detrimental. An examination of the assembly code for the routing algorithms shows that all the delay slots can be filled by the compiler. The processor, therefore, is already operating at peak performance with this particular benchmark. If the pipeline were deepened and branches resolved in a later stage, prediction may be worth pursuing. Without any major structural changes, however, it will not provide any improvement at this point.

Appendix B

R2000 ADL Architecture Description

```
################################################################################
# Standard MIPS pipeline with convntional internal data forwarding             #
#                                                                              #
# Author: Soner Onder                                                          #
################################################################################

processor processor_0 highbit 31
begin

lilliput little_endian;
Machineid "mips";

shadow register
   _hi                      32,          # Division operation HI value.
   _lo                      33,          # Division operation LO value.
   linebreak                32,          # Used in monitor statment
   branch_instruction_addr 32,
   check_ex                  1,
   check_mem                 1,
   check_wb                  1,
   ex_has_it                 1,
   mem_has_it                1,
   wb_has_it                 1,
   target                   32,          # Used in the branch target computation.
   data_tmp                 32,          # Used in data transfers.
   ptemp                    32,          # A temporary value register.
   dummy                    32,          # SAA (Same as above)
   which                     2,          # For passing parameters to cop branch units.
   less                      1,
   equal                     1,
   unordered                 1;

shadow register file dtemp[2,32];

latch
    exception   1,
    is_branch   1,
    load_flag_e 1,
    load_flag_m 1;

################################################################################
# Stall categories                                                             #
################################################################################
stall category
     ext_ref,
     latency_f,        # Floating point latency.
     latency_d,        # Divide latency.
     latency_m,        # Multiply latency.
     fpaddfull,        # Floating add pipeline is full with long latency op.
     fpmulfull,        # Floating multiply pipeline is full with lo la op.
     mem_ic,
     mem_dc,
     fl_d_dep,
     ld_d_dep,
     float_cc;
```

```
##########################################################################
# Register files                                                         #
##########################################################################
register file fpr_tag [33,1] # [34 regs,1 bit each].
   F_tag_0    0, F_tag_1    1, F_tag_2    2, F_tag_3    3,
   F_tag_4    4, F_tag_5    5, F_tag_6    6, F_tag_7    7,
   F_tag_8    8, F_tag_9    9, F_tag_10  10, F_tag_11  11,
   F_tag_12  12, F_tag_13  13, F_tag_14  14, F_tag_15  15,
   F_tag_16  16, F_tag_17  17, F_tag_18  18, F_tag_19  19,
   F_tag_20  20, F_tag_21  21, F_tag_22  22, F_tag_23  23,
   F_tag_24  24, F_tag_25  25, F_tag_26  26, F_tag_27  27,
   F_tag_28  28, F_tag_29  29, F_tag_30  30, F_tag_31  31,
   F_tag_CpC 32;

register file fpr [33,32]  # [34 regs,32 bits each].
   $f0        0, $f1       1, $f2        2, $f3        3,
   $f4        4, $f5       5, $f6        6, $f7        7,
   $f8        8, $f9       9, $f10      10, $f11      11,
   $f12      12, $f13     13, $f14      14, $f15      15,
   $f16      16, $f17     17, $f18      18, $f19      19,
   $f20      20, $f21     21, $f22      22, $f23      23,
   $f24      24, $f25     25, $f26      26, $f27      27,
   $f28      28, $f29     29, $f30      30, $f31      31,
   $CpC      32;

register file gpr [34,32]  # [34 regs,32 bits each].
   $0        0, $1        1, $2        2, $3         3,
   $4        4, $5        5, $6        6, $7         7,
   $8        8, $9        9, $10      10, $11       11,
   $12      12, $13      13, $14      14, $15       15,
   $16      16, $17      17, $18      18, $19       19,
   $20      20, $21      21, $22      22, $23       23,
   $24      24, $25      25, $26      26, $27       27,
   $28      28, $29      29, $30      30, $31       31,
   $zero     0, $at       1, $v0       2, $v1        3,
   $a0       4, $a1       5, $a2       6, $a3        7,
   $t0       8, $t1       9, $t2      10, $t3       11,
   $t4      12, $t5      13, $t6      14, $t7       15,
   $s0      16, $s1      17, $s2      18, $s3       19,
   $s4      20, $s5      21, $s6      22, $s7       23,
   $t8      24, $t9      25, $k0      26, $k1       27,
   $gp      28, $sp      29, $fp      30, $ra       31;

register file gpr_tag [34,32]  # [34 regs,32 bits each].
   gpr_tag_0   0, gpr_tag_1   1, gpr_tag_2   2, gpr_tag_3   3,
   gpr_tag_4   4, gpr_tag_5   5, gpr_tag_6   6, gpr_tag_7   7,
   gpr_tag_8   8, gpr_tag_9   9, gpr_tag_10 10, gpr_tag_11 11,
   gpr_tag_12 12, gpr_tag_13 13, gpr_tag_14 14, gpr_tag_15 15,
   gpr_tag_16 16, gpr_tag_17 17, gpr_tag_18 18, gpr_tag_19 19,
   gpr_tag_20 20, gpr_tag_21 21, gpr_tag_22 22, gpr_tag_23 23,
   gpr_tag_24 24, gpr_tag_25 25, gpr_tag_26 26, gpr_tag_27 27,
   gpr_tag_28 28, gpr_tag_29 29, gpr_tag_30 30, gpr_tag_31 31,
   gpr_tag_hi 32, gpr_tag_lo 33;


##########################################################################
# Memory port declarations                                               #
##########################################################################
memory mem2 latency  0 width 32;
memory icache latency 0 width 32;
memory dcache latency 0 width 32;


##########################################################################
# Pipeline, IR, PC, and controldata declarations                         #
##########################################################################
pipeline IPIPE       (s_IF, s_ID, s_EX, s_MEM, s_WB);
pipeline FP_ADD      (f_add1, f_add2, f_add3);
pipeline FP_MULTIPLY (f_mul1, f_mul2, f_mul3);

source s_IF;

instruction register ir 32;
instruction pointer  pc 32;

shadow register
       hi_val     32,
       lo_val     32;

latch
```

```
   new_pc          32,
   branch_input     1,
   branch_target   32;

controldata register
   my_pc        32,
   ls_bypass     1,
   mem_stat      1,
   access_type  32,
   byte          2,
   lop_r         6,     # lop_r indicates the register number for the lop.
   rop_r         6,     # rop_r indicates the register number for the rop.
   dest_r        6,     # dest_r holds the register number to write.
   simm         32,     # Sign extended immediate.
   zimm         32,     # Zero extended immediate.
   smdr         32,
   store_v      32,     # Store Memory data register.
   lmar         32,     # load memory address register.
   smar         32,     # store memory address register.
   dest         32,     # dest holds the value to be written.
   dest2        32,     # dest holds the value to be written.
   lop          32,     # lop holds the left operand value.
   lop2         32,     #
   rop          32,     # rop holds the right operand value.
   rop2         32;     #

################################################################################
# Constants                                                                    #
################################################################################
constant machine_drained        1;
constant cpc_register_number   32;
constant lo_hi_register_number 32;

bitconstant
   _BYTE        0 0,
   _HALFWORD    0 1,
   _TRIPLEBYTE  1 0,
   _WORD        1 1;


$include instruction-set.adl
$include mips-calling-convention.adl


################################################################################
# Procedures used by pipeline stages. Perform internal data forwarding        #
################################################################################
procedure do_forwarding_to_id untyped
begin
   if has_context s_EX then
      check_ex=(dest_type [s_EX] == integer_register) |
               (dest_type [s_EX] == lo_hi_register)
   else
      check_ex=0;

   if has_context s_MEM then
      check_mem=(dest_type [s_MEM] == integer_register) |
                (dest_type [s_MEM] == lo_hi_register)
   else
      check_mem=0;

   if has_context s_WB then
      check_wb=(dest_type [s_WB] == integer_register) |
               (dest_type [s_WB] == lo_hi_register)
   else
      check_wb=0;

   if (lop_type == integer_register) |
      (lop_type == lo_hi_register)  then
      begin
         if check_ex then
            ex_has_it=(dest_r[s_EX] == lop_r)
         else
            ex_has_it=0;

         if check_mem then
            mem_has_it=(dest_r[s_MEM] == lop_r)
         else
            mem_has_it=0;

         if check_wb then
```

```
               wb_has_it=(dest_r[s_WB] == lop_r)
            else
               wb_has_it=0;

            if ex_has_it then
               lop=dest[s_EX]
            else
            if mem_has_it then
               lop=dest[s_MEM]
            else
            if wb_has_it then
               lop=dest[s_WB];
         end;

      if (rop_type == integer_register) |
         (rop_type == lo_hi_register)  then
         begin
            if check_ex then
               ex_has_it=(dest_r[s_EX] == rop_r)
            else
               ex_has_it=0;

            if check_mem then
               mem_has_it=(dest_r[s_MEM] == rop_r)
            else
               mem_has_it=0;

            if check_wb then
               wb_has_it=(dest_r[s_WB] == rop_r)
            else
               wb_has_it=0;

            if ex_has_it then
               rop=dest[s_EX]
            else
            if mem_has_it then
               rop=dest[s_MEM]
            else
            if wb_has_it then
               rop=dest[s_WB];
         end;
   end do_forwarding_to_id;


   procedure do_forwarding_to_ex untyped
   begin
      if has_context s_MEM then
         check_mem=(dest_type [s_MEM] == integer_register) |
                   (dest_type [s_MEM] == lo_hi_register)
      else
         check_mem=0;

      if has_context s_WB then
         check_wb=(dest_type [s_WB] == integer_register) |
                  (dest_type [s_WB] == lo_hi_register)
      else
         check_wb=0;

      if (lop_type == integer_register) |
         (lop_type == lo_hi_register)  then
         begin
            if check_mem then
               mem_has_it=(dest_r[s_MEM] == lop_r)
            else
               mem_has_it=0;

            if check_wb then
               wb_has_it=(dest_r[s_WB] == lop_r)
            else
               wb_has_it=0;

            if mem_has_it then
               lop=dest[s_MEM]
            else
            if wb_has_it then
               lop=dest[s_WB];
         end;

      if (rop_type == integer_register) |
         (rop_type == lo_hi_register)  then
         begin
         if check_mem then
            mem_has_it=(dest_r[s_MEM] == rop_r)
         else
            mem_has_it=0;

         if check_wb then
            wb_has_it=(dest_r[s_WB] == rop_r)
         else
            wb_has_it=0;

         if mem_has_it then
            rop=dest[s_MEM]
         else
         if wb_has_it then
            rop=dest[s_WB];
      end;
end do_forwarding_to_ex;

integer waiting_drain;

################################################################################
# Instruction fetch stage                                                      #
################################################################################
procedure s_IF prologue
begin
   my_pc = pc;
   ir = icache[pc];

   if waiting_drain then
      begin
         waiting_drain =  has_context s_ID   +
                          has_context s_EX   +
                          has_context s_MEM  +
                          has_context s_WB   +
                          has_context f_add1 +
                          has_context f_add2 +
                          has_context f_add3 +
                          has_context f_mul1 +
                          has_context f_mul2 +
                          has_context f_mul3;

         if waiting_drain then
            stall ext_ref;

         builtin fast_va_start();
         builtin Fast_call_ext_reference(pc);
         pc = gpr[31];
         my_pc = pc;
         ir = icache[pc];
      end;

   if builtin is_external(pc) then
      begin
         waiting_drain = true;
         #builtin hold_the_pipeline();
         stall ext_ref;
      end;

   if access_complete then
      begin
         new_pc = pc+4;
         unfreeze;
      end
   else
      begin
         freeze;
         stall mem_ic;
      end;
end s_IF;

procedure s_IF epilogue
begin
   if send_enabled(s_ID) then
      begin
         send s_ID;

         if (branch_input) then
            begin
               branch_input=0;
               pc=branch_target;
            end
         else
            pc=new_pc;

         newcontext;
      end;
   end s_IF;

##################################################################################
# Instruction Decode stage                                                      #
##################################################################################
procedure s_ID prologue
begin
   decode;
end s_ID;

procedure s_ID intermission
begin
   # Check for data hazards and read operands.

   dest_r = ordinal(dest_reg);


   case lop_type of
   begin
      integer_register :
                          lop_r=rs;
                          lop=gpr[lop_r];

      lo_hi_register    : lop_r=lo_hi_register_number;
                          lop=gpr[lop_r];
                          lop2=gpr[lop_r+1];

      cpc_register      :
      float_register    :
                          lop_r=fs;
                          if fpr_tag[lop_r] then
                             stall fl_d_dep;
                          lop =fpr[lop_r];

      double_register   :
                          lop_r=fs;
                          if (fpr_tag[lop_r] > 0) | (fpr_tag[lop_r+1] > 0) then
                             stall fl_d_dep;
                          lop =fpr[lop_r];
                          lop2=fpr[lop_r+1];
   end;


   case rop_type of
   begin
      integer_register :
                          rop_r=rt;
                          rop=gpr[rop_r];

      lo_hi_register    : rop_r=lo_hi_register_number;
                          rop=gpr[rop_r];
                          rop2=gpr[rop_r+1];

      cpc_register      :
      float_register    : rop_r=ft;
                          if fpr_tag[rop_r] then
                             stall fl_d_dep;
                          rop =fpr[rop_r];

      double_register   : rop_r=ft;
                          if (fpr_tag[rop_r] > 0) | (fpr_tag[rop_r+1] > 0) then
                             stall fl_d_dep;
                          rop =fpr[rop_r];
                          rop2=fpr[rop_r+1];
   end;

   if has_context s_EX then
      if dest_type [s_EX] == lo_hi_register then
         stall;

   if has_context s_MEM then
      if dest_type [s_MEM] == lo_hi_register then
         stall;

   if has_context s_WB then
      if dest_type [s_WB] == lo_hi_register then
```

```
            stall;

      is_branch=(i_type ==  branch_type_0) | (i_type ==  branch_type_1);

      if has_context s_EX then
         load_flag_e=(i_type[s_EX ] == load_type)
      else
         load_flag_e=0;

      if has_context s_MEM then
         load_flag_m=(i_type[s_MEM] == load_type)
      else
         load_flag_m=0;

      if lop_type ^= none then
         begin
            if has_context s_EX then
            if (dest_r[s_EX]  == lop_r) & (load_flag_e | is_branch) then
               stall ld_d_dep;

            if has_context s_MEM then
            if (dest_r[s_MEM] == lop_r) & (load_flag_m | is_branch) then
               stall ld_d_dep;
         end;

      if rop_type ^= none then
         begin
            if has_context s_EX  then
            if (dest_r[s_EX]  == rop_r) & (load_flag_e | is_branch) then
               stall ld_d_dep;

            if has_context s_MEM then
            if (dest_r[s_MEM] == rop_r) & (load_flag_m | is_branch) then
               stall ld_d_dep;
         end;

      if (i_type ==  branch_type_0) | (i_type ==  branch_type_1) then
         begin
            if c_what ^= condition_z then
               branch_target=my_pc + sign_extend_14(immediate);

            if i_class == float_class then
               begin
                  if F_tag_CpC then
                     begin
                        # builtin printf("Stall on CPC\n");
                        stall float_cc;
                     end;
                  branch_input=$CpC == tf;
                  branch_instruction_addr=my_pc;
               end
            else
               begin
                  do_forwarding_to_id;
                  condition_code(lop,rop);
                  branch_instruction_addr=my_pc;
               end;

         end;
   end s_ID;

   procedure s_ID epilogue
   begin
      if ((exu == integer_unit ) | (exu == load_unit) |(exu == store_unit)) &
         (send_enabled(s_EX) == 0) then
         stall;

      if (exu == f_add_unit) & (send_enabled(f_add1) == 0) then
         stall fpaddfull;

      if (exu == f_mul_unit) & (send_enabled(f_mul1) == 0) then
         stall fpmulfull;

      if dest_type == float_register then
         fpr_tag[dest_r]=my_pc
      else
      if dest_type == double_register then
         begin
            fpr_tag[dest_r]=my_pc;
            fpr_tag[dest_r+1]=my_pc;
         end;
      if exu == f_add_unit then
         send f_add1
      else
      if exu == f_mul_unit then
         send f_mul1
      else
         send s_EX;
   end s_ID;

   ################################################################################
   # Instruction Execute stage                                                    #
   ################################################################################
   procedure s_EX prologue
   begin
      do_forwarding_to_ex;
   end s_EX;

   procedure s_EX epilogue
   begin
      send s_MEM;
   end s_EX;

   ################################################################################
   # Memory access stage                                                          #
   ################################################################################
   procedure s_MEM prologue
   begin
   end s_MEM;

   procedure s_MEM epilogue
   begin
      if exu == load_unit then
         begin
            if mem_stat == 0 then
               stall;
         end;
      if exu == store_unit then
         begin
            dcache.(access_type) [smar] = smdr;
            if access_complete == 0 then
               stall;
         end;
      send s_WB;
   end s_MEM;

   ################################################################################
   # Writeback stage                                                              #
   ################################################################################
   procedure s_WB prologue
   begin
      case dest_type of
      begin
         lo_hi_register   :
                            gpr[dest_r]=dest;
                            gpr[dest_r+1]=dest2;

         integer_register : gpr[dest_r]=dest;

         cpc_register     :
         float_register   : fpr[dest_r]=dest;
                            fpr_tag[dest_r]=0;

         double_register  : fpr[dest_r]=dest;
                            fpr[dest_r+1]=dest2;
                            fpr_tag[dest_r]=0;
                            fpr_tag[dest_r+1]=0;
      end;
   end s_WB;

   procedure s_WB epilogue
   begin
      retire stat;
   end s_WB;


   ################################################################################
   # Floating point add pipeline                                                  #
   ################################################################################
   procedure f_add1 prologue
   begin
```

```
end f_add1;

procedure f_add1 epilogue
begin
   send f_add2;
end f_add1;

procedure f_add2 prologue
begin
end f_add2;

procedure f_add2 epilogue
begin
   send f_add3;
end f_add2;

procedure f_add3 prologue
begin
end f_add3;

procedure f_add3 epilogue
begin
   send s_wb;
end f_add3;

###############################################################################
# Floating point multiply pipeline                                            #
###############################################################################
procedure f_mul1 prologue
begin
end f_mul1;

procedure f_mul1 epilogue
begin
   send f_mul2;
end f_mul1;

procedure f_mul2 prologue
begin
end f_mul2;

procedure f_mul2 epilogue
begin
   send f_mul3;
end f_mul2;

procedure f_mul3 prologue
begin
end f_mul3;

procedure f_mul3 epilogue
begin
   send s_wb;
end f_mul3;


procedure boot_up untyped
begin
   forall gpr = 0;
   forall fpr_tag=0;
end boot_up;

initialization boot_up;

controlflow
   bc1f__, beq__, bgez__, bgezal__,
   bgtz__, blez__, bltz__,
   bltzal__, bne__, j__,jal__,
   jalr__, jr__;

instruction category integer_arithmetic
   add, addi, addiu, addu, and, andi__, div,
   divu, lui, mfhi, mflo, mult, multu,
   nor, or, ori, sll, sllv, slt, slti,
   sltiu, sltu, sra, srav, srl, srlv,
   sub, subu, xor,xori;

instruction category conditional_branch
   beq__, bgez__, bgezal__,
   bgtz__, blez__, bltz__,
   bltzal__, bne__;
```

```
instruction category other
   break;

instruction category unconditional_branch
   j__, jal__,
   jalr__, jr__;

instruction category load
   lb, lbu, lh, lhu,
   lw, lwl, lwr, lwc1__;


instruction category store
   sb, sh, sw, swl,
   swr, swc1__;

instruction category float_arithmetic
   "cvt.d.w", "cvt.d.s", "cvt.s.w", "cvt.s.d",
   "div.d", "div.s", "mul.s", "mul.d",
   "add.s", "add.d", "neg.s", "neg.d",
   "sub.s", "sub.d",  mfc1, mtc1,
   "c.cond.d", "c.cond.s", "abs.s", "abs.d",
   "mov.s", "mov.d", "trunc.w.s", "trunc.w.d";

instruction category float_conditional
    bc1f__,
    bc1t__;

###############################################################################
# Display in debugger                                                         #
###############################################################################
monitor
   $0,   $1,   $2,   $3,   $4,   $5,   $6,   $7,   $8,   $9,   $10,
   $11,  $12,  $13,  $14,  $15,  $16,  $17,  $18,  $19,  $20,  $21,
   $22,  $23,  $24,  $25,  $26,  $27,  $28,  $29,  $30,  $31,
   linebreak,
   linebreak,
   $f0   , $f1   , $f2   , $f3   , $f4   , $f5   , $f6   , $f7   ,
   $f8   , $f9   , $f10  , $f11  , $f12  , $f13  , $f14  , $f15  ,
   $f16  , $f17  , $f18  , $f19  , $f20  , $f21  , $f22  , $f23  ,
   $f24  , $f25  , $f26  , $f27  , $f28  , $f29  , $f30  , $f31  ,
   linebreak,
   linebreak,
   pc;

end;

$include simulator-assembler-supplements.adl
```