Complexity-Effective Superscalar Processors

by

Subbarao Palacharla

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1998

Abstract

The performance trade-off between hardware complexity and clock speed in the design of superscalar microarchitectures is first investigated. Using the results of this trade-off analysis, the thesis proposes and evaluates two new superscalar microarchitectures designed with the goal of achieving high performance by reducing complexity.

This thesis takes a step towards quantifying the complexity of superscalar microarchitectures. First, a generic superscalar pipeline is defined. Then the specific areas of register renaming, instruction window wakeup, instruction window selection, register file access, and operand bypassing are analyzed. Each is modeled and Spice simulated for three different feature sizes representing past, present, and future technologies. Performance results and complexity trends are expressed in terms of issue width and window size. Results show that instruction window logic and operand bypass logic are likely to be the most critical in the future.

Following the complexity analysis, we study a family of superscalar microarchitectures called the dependence-based microarchitectures. These microarchitectures exploit natural dependences occurring in programs to reduce the complexity of window logic and operand bypass logic. Simulation results show that dependence-based superscalar microarchitectures are capable of extracting similar levels of parallelism as a conventional microarchitecture while facilitating a faster clock.

Finally, we propose and evaluate the integer-decoupled microarchitecture that improves the performance of integer programs by minimally adding to a conventional microarchitecture. Floating-point units in the conventional microarchitecture are augmented to perform simple integer operations and the resulting floating-point subsystem is used to

support some of the computation in integer programs. Simulation results are presented that show modest speedups for a 4-way processor. The speedups are attractive, however, considering that the proposed microarchitecture requires little additional hardware.

Acknowledgments

First and foremost, I thank my parents and family for encouragement and support during the seemingly endless stay in graduate school. I thank my dad for nudging me towards graduate school and research.

I am indebted to Jim Smith, my advisor, for taking me as his student at a crucial juncture in my graduate school career. I thank him for providing direction and for sharing his ideas. Most of all, I enjoyed his style of "loosely-coupled" advising. I also thank him for gladly answering my questions during countless walk-in meetings.

I also owe a lot to Norm Jouppi for helping me technically with the core of this dissertation. He patiently answered my questions, some stupid ones too, about VLSI circuits. I enjoyed many informative discussions about circuits and computer architecture with him. His advice and help made this thesis possible.

I am especially grateful to Guri Sohi for providing me with an office and computing facilities in Computer Sciences. I thank him for serving as a reader and for his critical comments on the thesis.

I thank David Wood, Jim Goodman, and Charles Kime for serving on my committee. I would like to especially thank David Wood for serving as a reader while on sabbatical and for making numerous useful comments that greatly improved the presentation of the thesis.

Over the past six years, I have had the privilege of technically interacting with Jim Goodman, Mark Hill, Rick Kessler, Norm Jouppi, Jim Smith, Guri Sohi, and David Wood. I thank them for teaching me most of what I know about computer architecture.

This dissertation has benefited from the work of other graduate students. Subramanya Sastry implemented the compiler support for a part of the dissertation research. Todd Austin developed the toolset on which the simulators used in this dissertation are based on. Alain Kägi always made time for helping me with Framemaker. I thank all of them.

I thank Scott Breach, Douglas Burger, Satish Chandra, Babak Falsafi, Alain Kägi, and T. N. Vijaykumar and for their friendship and company. Their camaraderie made life enjoyable and helped insulate me from occasional low points in life. Special thanks to Satish for the innumerable trips to State Street. Thanks to Amir Roth for fun discussions about anything and everything during the last year of my graduate studies. Outside of work, I thank Ambuj Shatdal, Francis Valiyaveetil, and Jignesh Patel for their company.

Finally, I would like to thank the agencies that funded my graduate studies. My work was supported in part by Wisconsin Alumni Research Foundation, NSF grants MIP-9505853, University of Wisconsin Graduate School, the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, Cray Research Inc., and Digital Equipment Corporation - Western Research Laboratory.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past decade superscalar microprocessors have become a source of tremendous computing power. They form the core of a wide spectrum of high-performance computer systems ranging from desktop computers to small-scale parallel servers to massively-parallel systems. To satisfy the ever-growing need for higher levels of computing power, computer architects need to investigate techniques that continue improving the performance of superscalar microprocessors while considering both changing technology and applications.

Superscalar microarchitectures [Joh91, SS95], on which superscalar microprocessors are based, deliver high performance by executing multiple instructions in parallel every cycle. Hardware is used to detect and execute parallel instructions. This technique of exploiting fine-grain parallelism at the instruction level to improve performance is commonly referred to as *instruction-level parallelism*. The maximum number of instructions processed in parallel, also known as the *width* of the microarchitecture, is typically four for the fastest microprocessors [Gwe96a, Kum96] available today. A typical superscalar

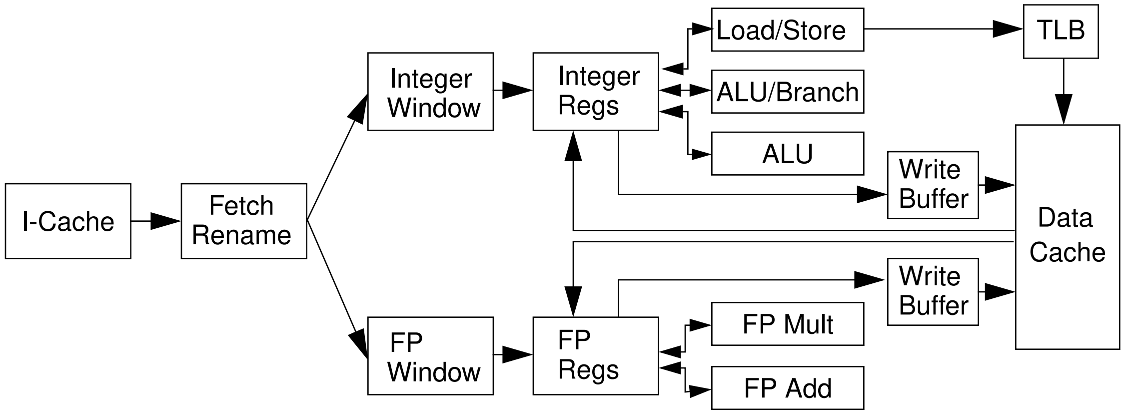


Figure 1-1. A typical superscalar microarchitecture.

microarchitecture, illustrated in Figure 1-1, operates as follows. Multiple instructions are fetched from the instruction cache every cycle. The instructions are then decoded, checked for dependences, renamed, and deposited in an instruction window. The instructions wait in the instruction window for their operands and functional units to become available. Hardware continuously monitors the dependences between instructions in the window and selects appropriate instructions for parallel execution. The overall hardware apparatus responsible for creating the window, monitoring dependences between instructions in the window, selecting instructions for execution from the window, and providing data operands to the instructions, henceforth collectively referred to simply as *issue logic*, is one of the most performance-critical components in a superscalar processor. The issue logic largely determines the amount of instruction-level parallelism that can be extracted. Hence, optimizing this logic is of paramount importance.

The net performance of a superscalar microarchitecture is directly proportional to the product —$Instructions\ Per\ Cycle \times Clock\ Frequency$. Instructions Per Cycle or IPC is the sustained number of instructions executed in parallel every cycle. IPC depends on a number of factors including the inherent parallelism in the program, the width of the microarchitecture, the size of the instruction window, and other characteristics of the scheme used for extracting parallelism. Clock Frequency is the speed at which the microarchitecture is clocked and is determined by the delays associated with the significant critical paths in the microarchitecture.

For the past decade, the general approach for improving the performance of superscalar microprocessors has been to build microarchitectures with increasingly complex issue logic that can boost the IPC factor in the performance equation. The increase in complexity results from a wider microarchitecture, a bigger instruction window, and more complex issue methods. However, there is a potential problem with continuing this strategy. While complex issue logic might be able to extract more parallelism, it can easily limit the clock speed of the microarchitecture. Microarchitectures with more complex issue logic typically require longer wires and deeper levels of logic to implement, and hence, can require longer critical paths in the microarchitecture. Thus, there is a danger of squandering the gains in IPC to a slow clock, resulting in reduced benefits or even no benefit in overall performance. Furthermore, technology trends suggest that wire delays will increasingly dominate total delay as feature sizes are reduced. These factors suggest that straightforward scaling of current microarchitectures for higher IPCs might not be the most appropriate approach for delivering higher performance in future. In summary, there is a trade-off between issue logic complexity, instructions per cycle (IPC), and clock speed that needs to be carefully examined while designing improved superscalar microarchitectures. This thesis examines this trade-off.

The above discussion underscores the need for investigating superscalar microarchitectures that judiciously use hardware complexity for exploiting significant levels of instruction-level parallelism while permitting a fast clock. We call such microarchitectures *complexity-effective superscalar microarchitectures*. These microarchitectures attempt to maximize the product of IPC and Clock Frequency rather than push the envelope for each term separately. This thesis proposes and evaluates two such complexity-effective superscalar microarchitectures called *dependence-based microarchitectures* and *integer-decoupled microarchitectures*.

It must be mentioned that the complexity of a design can have different, sometimes conflicting meanings. To a verification engineer, design A is more complex than design B if the time taken to verify design A is greater than that for design B. On the other hand, a logic designer typically measures complexity in terms of the number of gates required to implement a design. In this thesis, complexity is measured as the delay of the critical path through a piece of logic, and the longest path through any of the pipeline stages determines the clock speed. Complexity, as we define it, is largely independent of the number of gates required or the time to verify the design. Instead, complexity is dependent on a number of factors that could affect the delay of the critical paths in the design such as the number of logic stages, the length of wires, the degree of fan-out of a particular signal, and the number of associative compares performed every cycle.

While designing for complexity-effectiveness is a desirable goal, the question that immediately arises is: how do we quantify the complexity of a microarchitecture? It is commonplace to measure the IPC of a new microarchitecture, typically by using simulation. Such simulations count clock cycles and provide IPC in a direct manner. However, the complexity of a microarchitecture is much more difficult to determine — to be very accurate, it requires a full implementation in a specific technology. What is very much needed are fairly straightforward measures of complexity that can be used by microarchitects at a fairly early stage of the design process. Such methods would allow the determination of complexity-effectiveness. This thesis takes a step in the direction of characterizing complexity and complexity trends.

1.2 Historical Perspective

This section briefly outlines the evolution of ILP processors, especially superscalar processors, while highlighting major trends in design trade-offs involving hardware complexity and performance. Figure 1-2 illustrates the evolution of ILP processors with a time line.

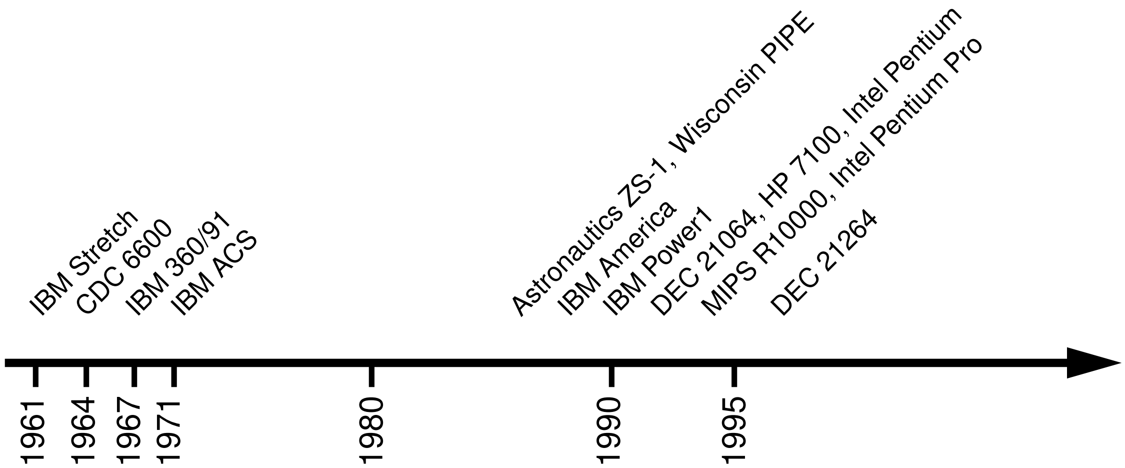


Figure 1-2. Time line showing evolution of superscalar processors.

Pipelining [Kog81] is the most prevalent technique for exploiting instruction-level parallelism. Pipelining enables overlapped execution of multiple instructions by breaking instruction processing into segments, just like an assembly line. It was first implemented in the IBM Stretch [Buc62] in 1961. Ever since, pipelining has been adopted by almost all high-performance designs.

The 1960s saw two pioneering machines that laid the foundation for much of the ILP techniques in wide use today. These were the CDC 6600 [Tho61,Tho63] and the IBM 360/91 [AST67] machines delivered in 1964 and 1967 respectively. The CDC 6600 implemented an impressive repertoire of architectural techniques, especially for its time — a clean load/store instruction set that enabled efficient pipelining, multiple functional units, and scoreboarding logic for dynamic scheduling. In the IBM 360/91 floating-point subsystem, the designers implemented a more sophisticated issuing scheme known as *Tomasulo's algorithm* [Tom67] after its inventor. The issuing schemes of most current superscalar microprocessors can be viewed as variants of Tomasulo's scheme. Even though the two designs implemented out-of-order execution, they were both single issue machines. Out-of-order execution was used to overlap execution of long-latency operations, tolerate slow memory accesses, and, in the case of the 360/91, mitigate the performance drawbacks of having few (8) floating-point registers.

Soon after, both IBM and CDC reverted back to simpler in-order issue, pipelined machines with a fast clock. The follow-on machines, the CDC 7600 and the IBM 360/85, issued instructions strictly in order. The exact reasons for this reversal are not known, but issues like the difficulty of debugging complex issue methods and the extra hardware cost are likely considerations on which the decision was based. Also, the use of a cache in the IBM 360/85 to tolerate memory latency probably made out-of-order execution less attractive. Two decades later, mushrooming transistor budgets, advanced CAD tools, and the market for high-performance, would trigger the resurgence of 6600 and 360/91-like schemes in the context of superscalar microprocessors.

The 1970s was not an eventful decade for ILP processors. All commercial machines still had a peak fetch rate of one instruction per cycle. However, during this time, some of the initial research in the area of multiple-instruction issue [TF70,RF72,Sch71] was carried out. Schorr describes an exploratory design [Sch71] capable of fetching, decoding, and executing multiple instructions every cycle. The design, later to be known as the IBM ACS (Advanced Computer System), was partitioned into the *index* unit that performed addressing operations and the *arithmetic* unit that executed arithmetic instructions. The arithmetic unit had a window of eight instructions out of which three instructions could be issued for execution every cycle. Unfortunately, the project was cancelled due to the incompatibility of the ISA with the S/360 ISA and other problems.

The late 1970s saw the emergence of a new paradigm for ILP called VLIW — Very Long Instruction Word — that grew out of early microcode machines [Wil51] and systems built by Floating Point Systems [Cha81]. VLIWs rely on the compiler to pack independent operations into a long instruction word which are then executed on multiple, independent functional units. The arguments in favor of VLIW are two-fold. First, since the compiler has a larger scope than the hardware to look for independent operations, VLIWs should be able to exploit more parallelism than superscalars. Second, since complex issue hardware is no longer required, VLIW processors can be clocked much faster than superscalar processors. However, even though a few commercial VLIW processors were built, the paradigm has not gained widespread acceptance. There are a number of reasons. First, to match hardware techniques, the paradigm requires sophisticated compiler technology that implements advanced techniques like software pipelining, global scheduling to move instructions across branches, trace scheduling [Fis81], and memory disambiguation. While advanced VLIW compilers [Ell85] that focussed on floating-point codes have been developed, it is not clear how well they perform on integer code where branches occur frequently and memory disambiguation is hard. Second, exposing hardware details to the compiler results in binaries that might not be portable across implementations. Third, the sophisticated transformations tend to result in increases in code size that can potentially degrade overall performance.

The lack of ILP innovation continued into the early 1980s. This was the period when most microprocessor designers were busy implementing RISC concepts [PS81] in the form of simple pipelining, and new ILP techniques did not receive much attention. However, the second half of the 1980s saw renewed ILP activity both in the superscalar and VLIW areas. The commercial implementations of the VLIW concept — Trace [CNO+88] by Multiflow and Cydra 5 [RYYT89] by Cydrome — were delivered during this time. However, these implementations had limited success in penetrating commercial markets. At the same time, three experimental superscalar prototype [S+87,GHL+85,Gro90] efforts were underway. These were the Astronautics ZS-1, the Wisconsin PIPE, and the IBM America machines. All three of them, implemented a limited form of multiple issue — integer instructions, including memory access related instructions, were issued in parallel with floating-point instructions. The ZS-1 and the PIPE used architectural queues to communicate values between the two classes of instructions. The America design used register renaming to achieve the same effect. All the designs still used in-order issue to execute instructions within each class. This simplified issue logic while allowing a limited form of out-of-order execution.

The early 1990s saw a number of superscalar implementations [KM89, D+92,K+93,Hsu94] — Intel i860, DEC 21064, HP 7100, MIPS R8000, and others. All of them, with the exception of the Power1, were simple in-order implementations that achieved multiple-issue by executing instructions of different types (load/store, branch, floating-point) in parallel. The IBM Power1 [Gro90] based on the earlier America design implemented register renaming and sophisticated instruction fetch mechanisms. Other vendors continued on the path of simple in-order implementations with a faster clock. This gave rise to the "speed demons" (simple implementations with a fast clock) versus "brainiacs" (complex implementations with a slow clock) controversy [Gwe93].

The mid 1990s saw some convergence between the two camps. Almost all vendors moved towards designs implementing complex out-of-order microarchitectures based on the 6600 and 360/91 schemes as well as ideas explored in academia [SP88,Soh90,HP86,DT92,YP92]. At the time of the writing of this thesis, every major microprocessor vendor has a product implementing sophisticated dynamic scheduling.

In 1996, Digital Equipment Corporation, long considered to be the bastion of the speed demons, announced plans for a product (DEC 21264 [Gwe96a]) implementing an out-of-order microarchitecture with a relatively fast clock (600 MHz). An interesting feature that stands out in this design is the microarchitectural changes employed to facilitate a fast clock. The integer subsystem is partitioned into two clusters. Instructions are steered from a central window to the clusters. Each cluster has its own copy of the register file. In addition to reducing the number of register file ports, clustering also makes possible fast bypassing between units in the same cluster. These features are described in more detail in Chapter 3. The research presented in this thesis has been highly influenced by this design.

In summary, the superscalar approach[1] has evolved over the years into the mainstream of processor implementations and each generation of designers had to deal with the trade-off between hardware complexity and performance.

1. There have been other ILP paradigms, some very successful in their own niche market, that have not been touched upon in this section. Some of these paradigms are vectors [Rus78], superpipelining [JW89], autotasking[ABHS89], multiprocessing[FJD80], and dataflow [DM74].

1.3 The Conventional Microarchitecture

As discussed earlier, current superscalar processors, like the MIPS R1000 [Yea96] and the DEC 21264 [Gwe96a], are typically based on the microarchitecture shown in Figure 1-1. The issue and execution resources in the machine are partitioned into integer and floating-point subsystems. The integer subsystem contains a number of load/store, branch, and functional units that operate on integer operands. The floating-point subsystem is similar to the integer subsystem except it does not contain load/store units, and it operates on floating-point operands. Instruction windows in each subsystem buffer instructions and implement dynamic scheduling as discussed earlier.

The microarchitecture presented in Figure 1-1 will be referred to as the conventional microarchitecture throughout the rest of this thesis. It will be used as a baseline for performance comparisons.

1.4 Thesis Contributions

1.4.1 Quantifying the Complexity of Superscalar Microarchitectures

The main contribution of this thesis is the development of simple models that both quantify the complexity of superscalar microarchitectures and identify complexity trends. Measurement of implementation complexity of microarchitectural features is going to be increasingly crucial for computer architects to understand and master. While much work remains to be done in this area, the work presented in this thesis is an important starting point.

The structures in a baseline superscalar microarchitecture whose complexity grows with increasing instruction-level parallelism are identified and analyzed. Each is modeled and Spice simulated for three different feature sizes representing past, present, and future technologies. Simple analytical models are developed that quantify the delay of these structures in terms of microarchitectural parameters of window size and issue width. The impact of technology trends towards smaller feature sizes is studied. In particular, the impact of poor scaling of wire delays in future technologies is analyzed.

In addition to delays, we study the performance effects of pipelining critical structures. Even if the delay of a structure is relatively large, it may not increase the complexity of the design because the structure's operation can be spread over multiple pipestages. Our analysis identifies structures that are more performance critical. The operation of these structures should be accommodated within a single cycle to avoid significant degradation in IPCs achieved, especially for programs with limited parallelism.

Our analysis shows that the issue window logic and data bypass logic are going to be the most critical structures in future. The delay of the issue window logic increases at least linearly with both issue width and window size. The functioning of this logic involves broadcasting of multiple tags on long wires spanning the window — an operation that does not scale well in future technologies. Furthermore, the delay of the window logic must fit in a pipestage to avoid performance degradation. Hence, this logic can be a key limiter of clock speed as we move towards wider issue widths, large window sizes, and advanced technologies in which wire delays dominate total delay. Another structure that can potentially limit clock speed especially in future technologies is the data bypass logic. The result wires that are used to bypass operand values increase in length as the number of functional units is increased. This results in a quadratic dependence of the bypass delay on issue width. Utilizing buffers helps mitigate the problem to an extent, but a linear increase in delay with issue width still persists. Just like the window logic, data bypass logic must also complete within a single cycle for performance reasons. Hence, bypass delays could ultimately become significant and force architects to consider more decentralized organizations.

1.4.2 Dependence-based Superscalar Microarchitectures

This thesis studies a new family of complexity-effective microarchitectures called dependence-based superscalar microarchitectures that address two major sources of complexity — window logic and data bypass logic — in conventional microarchitectures. Dependence-based microarchitectures use two main techniques to achieve the dual goals of high IPC and a fast clock. First, the machine is *partitioned* into multiple clusters each of which contains a slice of the instruction window and execution resources of the whole processor. This enables high-speed clocking of the clusters since the narrow issue width and the small instruction window of each cluster keeps critical delays small. The second technique involves *intelligent steering* of instructions to the multiple clusters so that the full width of the machine is utilized while minimizing the performance degradation due to slow inter-cluster communication.

A number of design alternatives and steering heuristics for dependence-based microarchitectures are proposed and evaluated using simulations. Among the designs presented, one that is particularly attractive is what we call the *fifo-based microarchitecture*. This microarchitecture implements the instruction window as a collection of a small number of fifos and steers dependent chains of instructions to the same fifo. Simulations show little slowdown as compared with a completely flexible issue window when performance is measured in clock cycles. Furthermore, because only instructions at fifo heads need to be awakened and selected, issue logic is simplified and the clock cycle is faster —consequently overall performance is improved. For example, our results show that, due to the clock speed advantage, the overall performance of a 2X4-way[1] fifo-based microarchitecture is 14% higher than that of a typical 8-way superscalar even though the proposed microarchitecture degrades IPC performance by 8% relative to the typical microarchitecture. By grouping dependent instructions together, the fifo-based microarchitecture also helps minimize the performance degradation due to slow bypasses in future wide-issue machines.

1. A 8-way microarchitecture comprising two clusters — each consisting of four fifos feeding four functional units.

1.4.3 Integer-decoupled Microarchitecture

This thesis proposes another complexity-effective microarchitecture called the *integer-decoupled microarchitecture* that improves the performance of integer programs and can be integrated into a conventional microarchitecture with little or no increase in complexity. The integer-decoupled microarchitecture starts with a conventional microarchitecture and augments the floating-point units to perform simple integer operations. Some integer instructions, those not used for computing addresses and accessing memory, are then off-loaded to the augmented floating-point subsystem by the compiler. Consequently, for integer programs, the integer-decoupled microarchitecture provides a larger window for dynamic scheduling as well as extra issue and execution bandwidth at no increase in complexity.

We evaluate the potential performance improvements with the integer-decoupled microarchitecture. Our results show that a modest to significant fraction of the total dynamic instructions in our benchmark programs can be off-loaded to the augmented floating-point subsystem. In doing so, the integer-decoupled microarchitecture provides speedups from 3% to 23% over a 4-wide (2 integer and 2 floating-point units) conventional microarchitecture. Furthermore, the results show that only simple integer operations need to be supported in the floating-point subsystem. This minimizes the additional hardware cost.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the simple models that we developed, along with the methodology used, for quantifying the complexity of superscalar microarchitectures. Chapter 3 proposes and evaluates dependence-based superscalar microarchitectures. Chapter 4 introduces and investigates the integer-decoupled microarchitecture. Finally, Chapter 5 gives conclusions and suggests future directions to explore. The appendices includes detailed experimental results for Chapter 2.

Chapter 2

Quantifying the Complexity of Superscalar Microarchitectures

The complexity of a microarchitecture is difficult to determine — to be very accurate, it would require a full implementation in a specific technology. What is very much needed are fairly straightforward measures, possibly only relative measures, of complexity that can be used by microarchitects at a fairly early stage of the design process. This chapter presents work that takes a step in that direction. Simple models that quantify the complexity of superscalar microarchitectures are developed and used to identify long-term complexity trends.

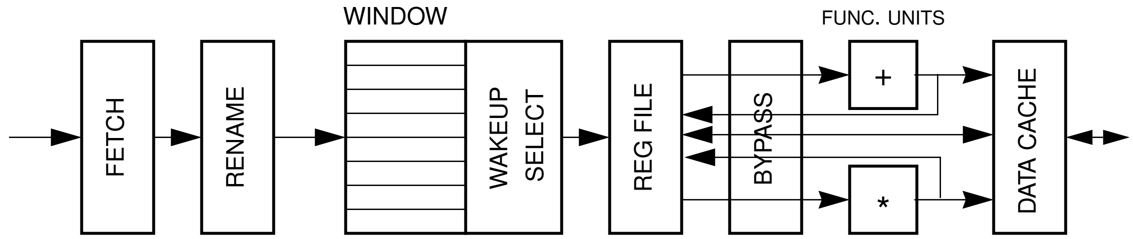
We start by identifying those portions of a microarchitecture whose complexity grows with increasing instruction-level parallelism. Of these, we focus on register rename logic, window logic, register file logic, and data bypass logic. We analyze potential critical paths in these structures and develop models for quantifying their delays. We study the manner in which these delays vary with microarchitectural parameters like window size (the number of instructions from which ready instructions are selected for issue) and issue width (the number of instructions that can be issued in a cycle). We also study the impact of the technology trend towards smaller feature sizes. In particular, we analyze how the poor scaling of wire delays in future affects the overall delay of critical structures.

In addition to delays, we study the performance effects of pipelining critical structures. Even if the delay of a structure is relatively large, it may not increase the complexity of the design because the structure's operation can be spread over multiple pipestages. We analyze structures to identify those whose operation must be accomplished within a single cycle to avoid significant degradation in the number of instructions committed every cycle.

The rest of this chapter is organized as follows. Section 2.1 describes the sources of complexity in a baseline microarchitecture. Section 2.2 describes the methodology we use to study the critical structures identified in Section 2.1. Section 2.3 briefly discusses technology trends. Section 2.4 presents a detailed analysis of each structure and how the delay of the structure varies with microarchitectural parameters and technology parameters. Section 2.5 discusses pipelining issues for each of the structures and presents overall delay results. Finally, Section 2.6 lists related work, and Section 2.7 summarizes the chapter.

2.1 Sources of Complexity

Before delving into specific sources of complexity, we describe the baseline superscalar model assumed for the study. We then list the basic structures that are the primary sources of complexity. Finally, we show how these basic structures are present in one form or another in most current implementations even though these implementations might appear to be different superficially. On the other hand, we realize that it is impossible to capture all possible microarchitectures in a single model and any results provided here have some obvious limitations. We can only provide a fairly straightforward model that is typical of most current superscalar processors, and suggest that techniques similar to those used here can be extended for other, more advanced models as they are developed.



FETCH	RENAME	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	COMMIT

Figure 2-1. Baseline superscalar model.

Figure 2-1 illustrates the baseline model and the associated pipeline. The fetch unit fetches multiple instructions every cycle from the instruction cache. Branches encountered by the fetch unit are predicted. Following instruction fetch, instructions are decoded and their register operands are renamed. Register renaming involves mapping the logical register operands of an instruction to the appropriate physical registers. Renamed instructions are then deposited in the issue window, where they wait for their source operands and the appropriate functional unit to become available. As soon as these conditions are satisfied, the instruction is issued and executes on one of the functional units. The operand values of an instruction are either fetched from the register file or are bypassed from earlier instructions in the pipeline. The data cache provides low latency access to memory operands via loads and stores.

The issue window is responsible for monitoring dependences between instructions in the window and issuing instructions to the functional units. The window logic consists of two components — the *wakeup* logic and the *select* logic. The first component is responsible for “waking up” instructions waiting in the issue window for their source operands to become available. Once an instruction is issued for execution, the tag corresponding to its result is broadcast to all the instructions in the window. Each instruction in the window compares the tag with its source operand tags. Once all the source operands of an instruction are available the instruction is flagged *ready* for execution. The select logic is responsible for selecting instructions for execution from the pool of ready instructions. An instruction is said to be ready if all of its source operands are available. As pointed out earlier, the wakeup logic is responsible for setting the ready flag.

2.1.1 Basic Structures

The most important criterion used for identifying a basic structure for our study is that the delay of the structure should be a function of either issue window size or issue width or both. For example, we consider register renaming to be a basic structure because its delay depends on the number of ports into the mapping table which in turn is determined by the issue width. On the other hand none of the functional units are included in the study because their delay is independent of both the issue width and the window size. In addition, our decision to study a particular structure was based on two observations. First, we are primarily interested in dispatch and issue-related structures because these structures form the core of a microarchitecture and largely determine the amount of parallelism that can be exploited. Second, some of these structures rely on broadcast operations on long wires and hence, their delays might not scale as well as logic-intensive structures in future technologies with smaller feature sizes. Hence, we believe that these structures are potential cycle-time determinants in future wide-issue designs in advanced technologies.

The structures we consider are:

- *Register rename logic*
- *Window wakeup logic*
- *Window selection logic*
- *Register file logic*
- *Data bypass logic*

There are other important pieces of logic that are not considered in this thesis, even though their delay is a function of issue width. These are:

- *Caches.*

Instruction and data caches provide low latency access to instructions and memory operands, respectively. In order to provide the necessary load/store bandwidth [SF91] in a superscalar processor, the cache has to be banked or duplicated. The access time of a cache is a function of the size of the cache and the associativity of the cache. Wada et al. [WRP92] and Wilton and Jouppi [WJ94] have developed detailed models that estimate the access time of a cache given its size and associativity.

- *Instruction fetch logic*

Besides the instruction cache, there are other important parts of fetch logic whose complexity varies with dispatch width. First of all, as instruction issue widths grow beyond the size of a single basic block, it will become necessary to predict multiple branches every cycle. Then, non-contiguous blocks of instructions will have to be fetched from the instruction cache and compacted into a contiguous block prior to renaming. Rotenberg et al. [RBS96] describe the logic required for these operations. However, delay models remain to be developed. And, although they are important, they are not considered here.

Finally, it must be pointed out once again that in real designs there may be structures not listed above that influence the overall delay of the critical path. However, our realistic aim is not to study all of them but to analyze in detail some important ones that have been reported in the literature. We believe that our basic technique can be applied to others, however.

2.1.2 Current Implementations

The structures identified above were presented in the context of the baseline superscalar model shown in Figure 2-1. The MIPS R10000 [Yea96], and the DEC 21264 [Gwe96a] are two implementations of this model. Hence, the structures identified above apply to these two processors.

On the other hand, the Intel Pentium Pro [Gwe95b], the PowerPC 604 [SDC95], and the HAL SPARC64 [Gwe95a] are based on the reservation model shown in Figure 2-2. There are two main differences between the two models. First, in the baseline model all the register values, both speculative and non-speculative, reside in the physical register file. In the reservation station model, the reorder buffer holds speculative values and the register file

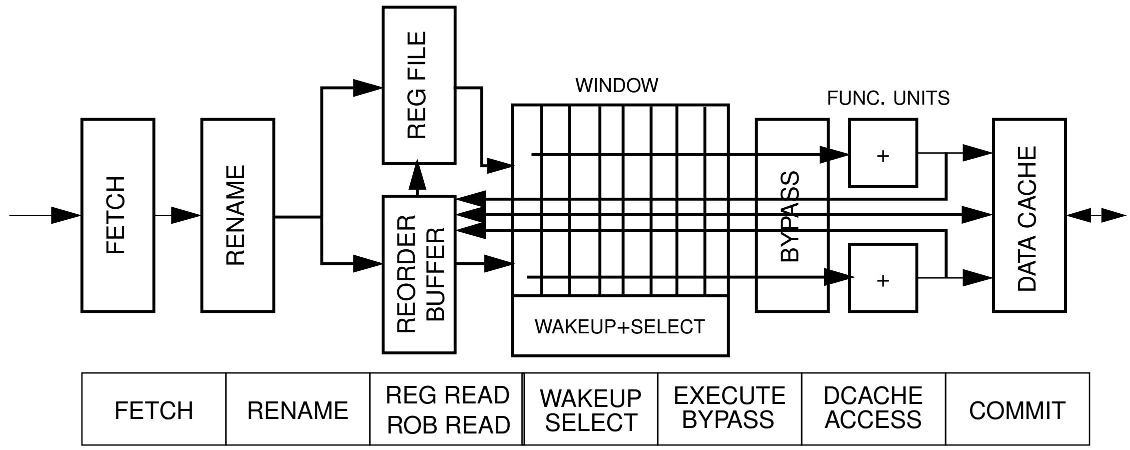


Figure 2-2. Reservation stations-based superscalar model.

holds only committed, non-speculative data. Second, operand values are not broadcast to the window entries in the baseline model - only their tags are broadcast; data values go to the physical register file. In the reservation station model, completing instructions broadcast result values to the reservation stations. Issuing instructions read their operand values from the reservation station.

The point to be noted is that the basic structures identified earlier are also present in the reservation station model and are as critical as in the baseline model. The only notable difference is that the reservation station model has a smaller physical register file (equal to the number of architected registers) and might not demand as much bandwidth (as many ports) as the register file in the baseline model, because in this case some of the operands come from the reorder buffer and the reservation stations.

While the discussion of potential sources of complexity is in the context of a baseline superscalar model that is out-of-order, it must be pointed out that some of the critical structures identified apply to in-order processors too. For example, the register file logic, and the data bypass logic are also present in in-order superscalar processors.

2.2 Methodology

Each structure was studied in two phases. In the first phase, a representative CMOS circuit was selected for the structure. This was done by studying designs published in the literature[1] and by collaborating with engineers at Digital Equipment Corporation. In cases where there was more than one possible design, we performed a preliminary study of the designs to select one that was most promising. In one case, register renaming, we had to study (simulate) two different schemes.

In the second phase, the circuit was implemented and optimized for speed. Circuits were designed mostly using static logic. We believe that power and robustness considerations will make static logic more attractive than dynamic logic in future. However, in situations where dynamic logic helped boost the performance significantly, dynamic logic was used. For example, in the window wakeup logic, a dynamic 7-input NOR gate was used for comparisons instead of a static gate. A number of optimizations were applied to improve the speed of the circuits. First, all the transistors in the circuit were manually sized so that overall delay improved. Second, logic optimizations like two-level decomposition were applied to reduce fan-in requirements. Static gates with a fan-in greater than four were avoided. Third, in some cases transistor reordering was used to shorten the critical path. Some of the optimization sites will be pointed out when the individual circuits are described.

We used the HSPICE circuit simulator [Met87] from MetaSoftware to simulate the circuits. In order to simulate the effect of wire parasitics, parasitics were added at appropriate nodes in the Hspice model of the circuit. These parasitics were computed by calculating the length of the wires based on the layout of the circuit and using the values of R_{metal} and C_{metal} — the resistance and parasitic capacitance of metal wires per unit length.

1. Mainly proceedings of the ISSCC — International Solid-State and Circuits Conference.

To study the effect of reducing the feature size on the delays of the structures, we simulated the circuits for three different feature sizes: 0.8µm, 0.35µm, and 0.18µm respectively. The process parameters for the 0.8µm CMOS process were taken from Johnson and Jouppi's synthetic model [JJ90]. These parameters were used by Wilton and Jouppi [WJ94] to study the access time of caches. Because process parameters are proprietary information, we had to use extrapolation to come up with process parameters for the 0.35µm and 0.18µm technologies. We used the 0.8µm process parameters from Johnson and Jouppi's synthetic model [JJ90], 0.5µm process parameters from MOSIS, and process parameters used in the literature as inputs. The process parameters assumed for the three technologies are listed in Appendix A. Layouts for the 0.35µm and 0.18µm technologies were obtained by appropriately shrinking the layout for the 0.8µm technology.

Finally, basic RC circuit analysis was used to develop simple analytical models that captured the dependence of the delays on microarchitectural parameters like issue width and window size. The relationships predicted by the Hspice simulations were compared against those predicted by our model. In most of the cases, our models were accurate in identifying the relationships.

2.2.1 Caveats

The above methodology does not address the issue of how well the assumed circuits reflect real circuits for the structures. However, by basing our circuits on designs published by microprocessor vendors, we believe that the assumed circuits are close to real circuits. In practice, many circuit tricks can be employed to optimize critical paths for speed. However, we believe that the relative delay times between different configurations should be more accurate than the absolute delay times. Because we are mainly interested in finding trends in the manner in which delays of the structures vary with microarchitectural parameters like window size and issue width, and how the delays scale as the feature size is reduced, we believe that our results are valid.

It must also be pointed out that while the absolute delay times presented in this thesis track the resulting clock speed, they cannot be directly converted into clock speeds. There are two reasons for this. First, we do not include the delay of inter-stage latches and the delay resulting from clock skew in our measurements. These two components can be responsible for a non-trivial fraction of the total delay [NH97], especially for high frequency designs. Second, the delay of a design can show considerable variance with process parameters and temperature of operation. Commercial designs are required to operate over a range of process parameters and physical temperatures. Our designs were simulated for a single set of process parameters and a single temperature point (25 °C).

2.2.2 Terminology

Table 2.1 defines some of the common terms used in the rest of this chapter. The remaining terms will be defined when they are introduced.

Symbol	Represents
IW	Issue width
$WINSIZE$	Window size
$NVREG$	Number of logical registers
$NPREG$	Number of physical registers
$NVREG_{width}$	Width of logical register tags
$NPREG_{width}$	Width of physical register tags
$DATA_{width}$	Width of datapath
R_{metal}	Resistance of metal wire per unit length
C_{metal}	Capacitance of metal wire per unit length

Table 2.1: Terminology.

2.3 Technology Trends

Feature sizes of MOS devices have been steadily decreasing. This trend [Ass97] towards smaller devices is likely to continue at least for the next decade. In this section, we briefly discuss the effect of shrinking feature sizes on circuit delays. The effect of scaling feature sizes on circuit performance is an active area of research [D+74, MF95]. We are only interested in illustrating the trends in this section.

Circuit delays consist of logic delays and wire delays. Logic delays result from gates that drive other gates. Wire delays are the delays resulting from driving values on wires.

2.3.1 Logic Delays

The delay of a logic gate can be written as

$$Delay_{gate} = (C_L \times V)/I$$

where C_L is the load capacitance at the output of the gate, V is the supply voltage, and I is the average charging/discharging current. I is a function of I_{dsat} — the saturation drain current of the devices forming the gate. As the feature size is reduced, the supply voltage has to be scaled down to keep the power consumption at manageable levels. Because voltages cannot be scaled arbitrarily they follow a different scaling curve from feature sizes. For submicron devices [Rab96], if S is the scaling factor for feature sizes, and U is the scaling factor for supply voltages, then C_L, V, and I scale by factors of $1/S$, $1/U$, and $1/U$ respectively. Hence, the overall gate delay scales by a factor of $1/S$. Therefore, gate delays decrease uniformly as the feature size is reduced.

2.3.2 Wire Delays

If L is the length of a wire, then the intrinsic RC delay of the wire is given by

$$Delay_{wire} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

where R_{metal}, C_{metal} are the resistance and parasitic capacitance of metal wires per unit length respectively and L is the length of the wire. The factor 0.5 is introduced because we use the first order approximation that the delay at the end of a distributed RC line is

$(RC)/2$ (we assume the resistance and capacitance are distributed uniformly over the length of the wire).

In order to study the impact of shrinking feature sizes on wire delays we first have to analyze how the resistance, R_{metal}, and the parasitic capacitance, C_{metal}, of metal wires vary with feature sizes. We use the simple model presented by Bohr [Boh95] to estimate how R_{metal} and C_{metal} scale with feature size. Note that both these quantities are per unit length measures. Using Bohr's model [Boh95],

$$
\begin{aligned}
R_{metal} &= \rho/(width \times thickness) \\
C_{metal} &= C_{fringe} + C_{parallelplate} \\
&= 2 \times \varepsilon \times \varepsilon_0 \times (thickness)/(width) + 2 \times \varepsilon \times \varepsilon_0 \times (width)/(thickness)
\end{aligned}
$$

where *width* is the width of the wire, *thickness* is the thickness of the wire, ρ is the resistivity of metal, and ε and ε_0 are permittivity constants.

The average metal thickness has remained relatively constant for the last few generations while the width has been decreasing in proportion to the feature size. Hence, if S is the scaling factor for feature sizes, the scaling factor for R_{metal} is S. The metal capacitance has two components: fringe capacitance and parallel-plate capacitance. Fringe capacitance is the result of capacitance between the side-walls of adjacent wires and capacitance between the side-walls of the wires and the substrate. Parallel-plate capacitance is the result of capacitance between the bottom-wall of the wires and the substrate. Assuming that the thickness remains constant, it can be seen from the equation for C_{metal} that the fringe capacitance becomes dominant as we move towards smaller feature sizes. Rahmat et al. [RNOM95] show that as feature sizes are reduced, the fringe capacitance will be responsible for an increasingly larger fraction of the total capacitance. For example, they show that for feature sizes less than 0.1μm, the fringe capacitance contributes 90% of the total capacitance. In order to accentuate the effect of wire delays and to be able to identify their effects, we assume that the metal capacitance is largely determined by the fringe capacitance and therefore the scaling factor for C_{metal} is also S.

Using the above scaling factors in the equation for the wire delay, we can compute the scaling factor for wire delays as,

$$\begin{aligned} \text{Scaling Factor} &= S \times S \times (1/S)^2 \\ &= 1 \end{aligned}$$

Note that the length scales as $1/S$ for local interconnects. In this study we are only interested in local interconnects. This might not be true for global interconnects like the clock because their length also depends on the die size.

Hence, as feature sizes are reduced, wire delays remain constant. This, coupled with the fact that logic delays decrease uniformly with feature size, implies that wire delays will dominate total delays in future. In reality, the situation is further aggravated for two reasons. First, not all wires reduce in length perfectly (by a factor of S). Second, some of the global wires, like the clock, actually increase in length due to bigger dice that are made possible with each generation.

McFarland and Flynn [MF95] studied various scaling schemes for local interconnect and conclude that a *quasi-ideal* scaling scheme closely tracks future deep submicron technologies. Quasi-ideal scaling performs ideal scaling of the horizontal dimensions but scales the thickness more slowly. The scaling factor for RC delay per unit length for their scaling model is $(0.9 \times S^{1.5} + 0.1 \times S^{2.5})$. In comparison, for our scaling model, the scaling factor for RC delay per unit length is a more conservative, and simpler, S^2.

2.4 Complexity Analysis

In this section we discuss the critical structures in detail. The presentation of each structure is organized as follows. First, we describe the logical function implemented by the structure. Then, we present possible schemes for implementing the structure and describe one of the schemes in detail. Next, we analyze the overall delay of the structure in terms of microarchitectural parameters like issue width and window size using simple delay models. Finally, we present Spice simulation results, identify trends in the results and discuss how the results conform to the delay analysis performed earlier.

2.4.1 Register Rename Logic

The register rename logic is used to translate logical register designators into physical register designators. Logically, this is accomplished by accessing a map table with the logical register designator as the index. Because multiple instructions, each with multiple register operands, need to be renamed every cycle, the map table has to be multi-ported. For example, a 4-wide issue machine with two read operands and one write operand per instruction requires 8 read ports and 4 write ports into the mapping table. The high level block diagram of the rename logic is shown in Figure 2-3. The map table holds the current logical to physical mappings. In addition to the map table, dependence check logic is required to detect cases where the logical register being renamed is written by an earlier instruction in the current group of instructions being renamed. The dependence check logic detects such dependences and sets up the output MUXes so that the appropriate physical register designators are generated. The shadow table is used to checkpoint old mappings so that the processor can quickly recover to a precise state from branch mispredictions. At the end of every rename operation, the map table is updated to reflect the new logical to physical mappings created for the result registers of the current rename group.

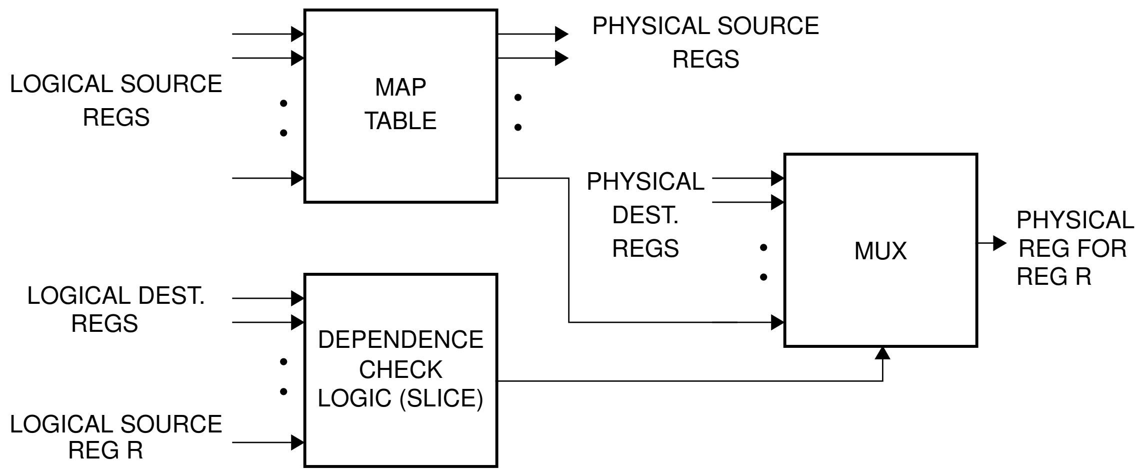


Figure 2-3. Register rename logic.

2.4.1.1 Structure

The mapping and checkpointing functions of the rename logic can be implemented in at least two ways. These two schemes, called the RAM scheme and the CAM scheme, are described next.

RAM scheme

In the RAM scheme, as implemented in the MIPS R10000 [Yea96], the map table is a RAM where each entry contains the physical register that is mapped to the logical register whose designator is used to index the table. The number of entries in the map table is equal to the number of logical registers. A single cell of the table is shown in Figure 2-5. A shift register, present in every cell, is used for checkpointing old mappings.

The map table works like a register file. The bits of the physical register designators are stored in the cross-coupled inverters in each cell. A read operation starts with the logical register designator being applied to the decoder. The decoder decodes the logical register designator and raises one of the word lines. This triggers bit line changes which are sensed by a sense amplifier and the appropriate output is generated. Precharged, double-ended bit lines are used to improve the speed of read operations. Mappings are checkpointed by copying the current contents of each cell into the shift register. Recovery is performed by writing the bit in the appropriate shift register cell back into the main cell.

CAM scheme

An alternative scheme for register renaming uses a CAM (content-addressable memory) to store the current mappings. Such a scheme is implemented in the HAL SPARC [AMG+95] and the DEC 21264 [Kel96]. The number of entries in the CAM is equal to the number of physical registers. Each entry contains two fields. The first field stores the logical register designator that is mapped to the physical register represented by the entry. The second field contains a valid bit that is set if the current mapping is valid. The valid bit is required because a single logical register designator might map to more than one physical register. When a mapping is changed, the logical register designator is written into the entry corresponding to a free physical register and the valid bit of the entry is set. At the same time, the valid bit used for the previous mapping is located through an associative search and cleared.

The rename operation in this scheme proceeds as follows. The CAM is associatively searched with the logical register designator. If there is a match and the valid bit is set, a read enable wordline corresponding to the CAM entry is activated. An encoder (ROM) is used to encode the read enable word lines (one per physical register) into a physical register designator. Old mappings are checkpointed by storing the valid bits from the CAM into a checkpoint RAM. To recover from an exception, the valid bits corresponding to the old mapping are loaded into the CAM from the checkpoint RAM. In the HAL design, up to 16 old mappings can be saved.

The CAM scheme is less scalable than the RAM scheme because the number of CAM entries, which is equal to the number of physical registers, increases with issue width. In order to support such a large number of physical registers, the CAM will have to be appropriately banked. On the other hand, in the RAM scheme, the number of entries in the map

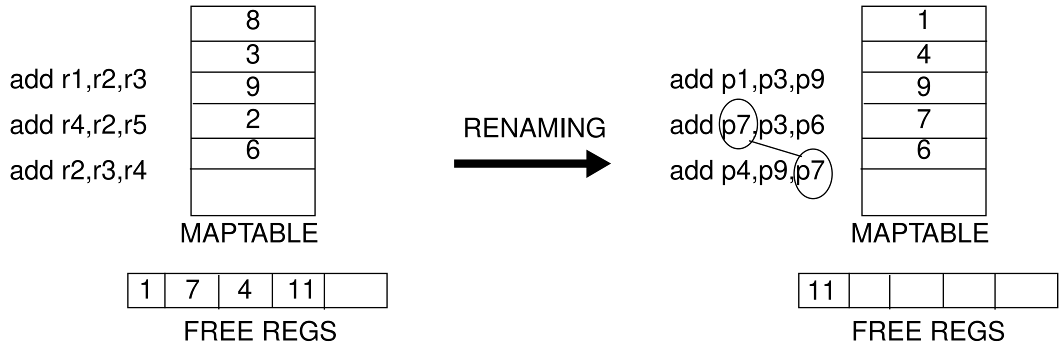


Figure 2-4. Renaming example showing dependency checking. The first entry of the map table corresponds to logical register r1.

table is independent of the number of physical registers. However, the CAM scheme has an advantage with respect to checkpointing. In order to checkpoint in the CAM scheme, only the valid bits have to be saved. This is easily implemented by having a RAM adjacent to the column of valid bits in the CAM. In other words, the dimensions of the individual CAM cells is independent of the number of checkpoints. On the other hand, in the RAM scheme, the width of individual cells is a function of the number of checkpoints because this number determines the length of the shift register in each cell.

The dependence check logic proceeds in parallel with the map table access. Every logical register designator being renamed is compared against the destination register designators (logical) of earlier instructions in the current rename group. If there is a match, then the tag corresponding to the physical register assigned to the earlier instruction is used instead of the tag read from the map table. For example, in the case shown in Figure 2-4, the last instruction's operand register r4 is mapped to p7 and not p2. In the case of more than one match, the tag corresponding to the latest (in dynamic order) match is used. We implemented the dependence check logic for issue widths of 2, 4, and 8. We found that for these issue widths, the delay of the dependence check logic is less than the delay of the map table, and hence the check can be hidden behind the map table access.

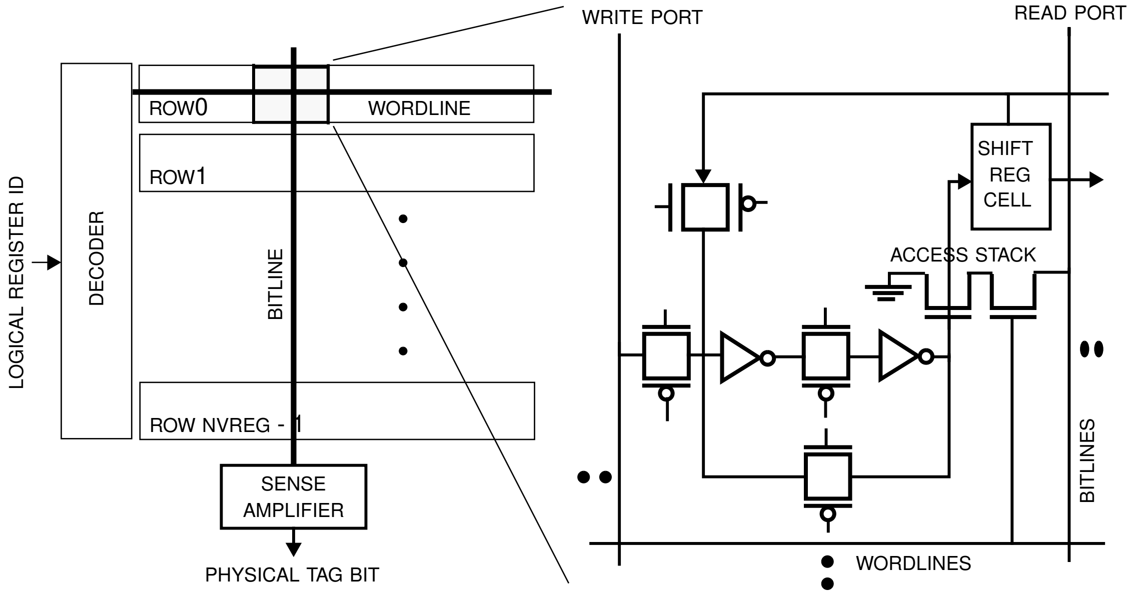


Figure 2-5. Rename map table. This figure shows the map table of the rename logic on the left and a single cell of the map table on the right.

2.4.1.2 Delay Analysis

We implemented both the RAM scheme and the CAM scheme. We found the performance of the two schemes to be comparable for the design space we explored. To keep the analysis short and since the RAM scheme is more scalable, we will only discuss the RAM scheme here.

A single cell of the map table is shown in Figure 2-5. The critical path for the rename logic is the time it takes for the bits of the physical register designator to be output after the logical register designator is applied to the address decoder. The delay of the critical path consists of three components: the time taken to decode the logical register designator, the time taken to drive the wordline, the time taken by an access stack to pull the bitline low plus the time taken by the sense amplifier to detect this bitline change and produce the corresponding output. The time taken for the output of the map table to pass through the MUX in Figure 2-3 is ignored because this is very small compared to the rest of the rename logic and, more importantly, the control input of the MUX is available in advance because the dependence check logic is faster than the map table. Hence, the overall delay is given by,

$$Delay = T_{decode} + T_{wordline} + T_{bitline}$$

Each of the components is analyzed next.

Decoder delay

The structure of the decoder is shown in Figure 2-6. We use predecoding to improve the speed of decoding. The predecode gates are 3-input NAND gates and the row decode gates are 3-input NOR gates. The output of the NAND gates is connected to the input of the NOR gates by the predecode lines. The length of these lines is given by,

$$PredeclineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NVREG$$

where *cellheight* is the height of the a single cell excluding the wordlines, *IW* is the issue width, $wordline_{spacing}$ is the spacing between wordlines, and *NVREG* is the number of logical registers. The factor 3 in the equation results from the assumption of 3-operand instructions (2 read operand and 1 write operand). With these assumptions, 3 ports (2 read ports and 1 write port) are required per cell for each instruction being renamed. Hence, for a IW-wide issue machine, a total of $3 \times IW$ wordlines are required for each cell

The decoder delay is the time it takes to decode the logical register designator i.e. the time it takes for the output of the NOR gate to rise after the input to the NAND gate has been applied. Hence, the decoder delay can be written as

$$T_{decode} = T_{nand} + T_{nor}$$

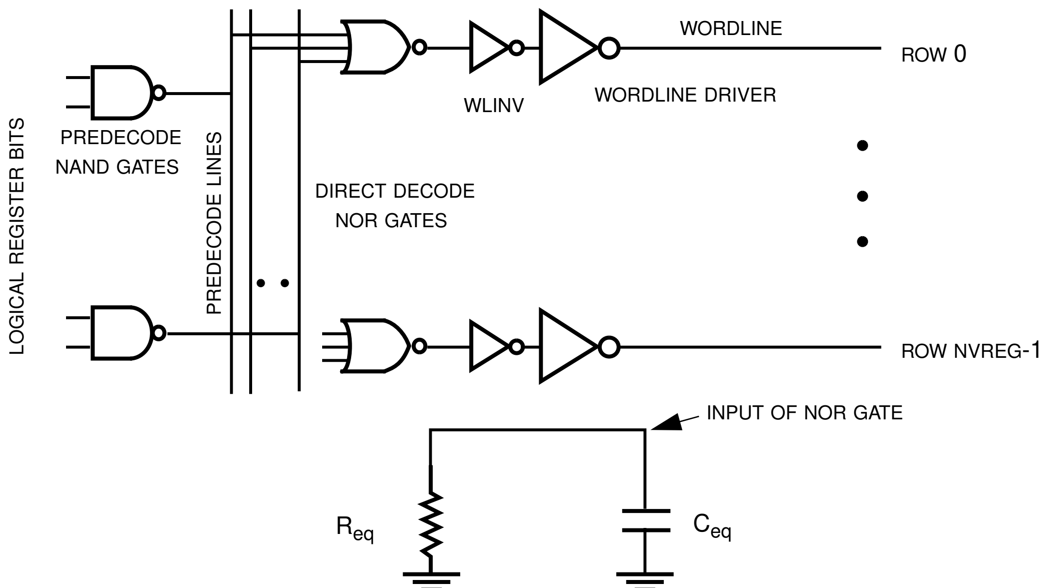


Figure 2-6. Decoder structure and equivalent circuit.

where T_{nand} is the delay of the NAND gate and T_{nor} is the delay of the NOR gate. From the equivalent circuit of the NAND gate shown in Figure 2-6.

$$T_{nand} = c_0 \times R_{eq} \times C_{eq}$$

R_{eq} consists of two components: the resistance of the NAND pull-down and the metal resistance of the predecode line connecting the NAND gate to the NOR gate. Hence,

$$R_{eq} = R_{nandpd} + 0.5 \times PredeclineLength \times R_{metal}$$

Note that we have divided the resistance of the predecode line by two; the first order approximation for the delay at the end of a distributed RC line is RC/2 (we assume that the resistance and capacitance are distributed evenly over the length of the wire).

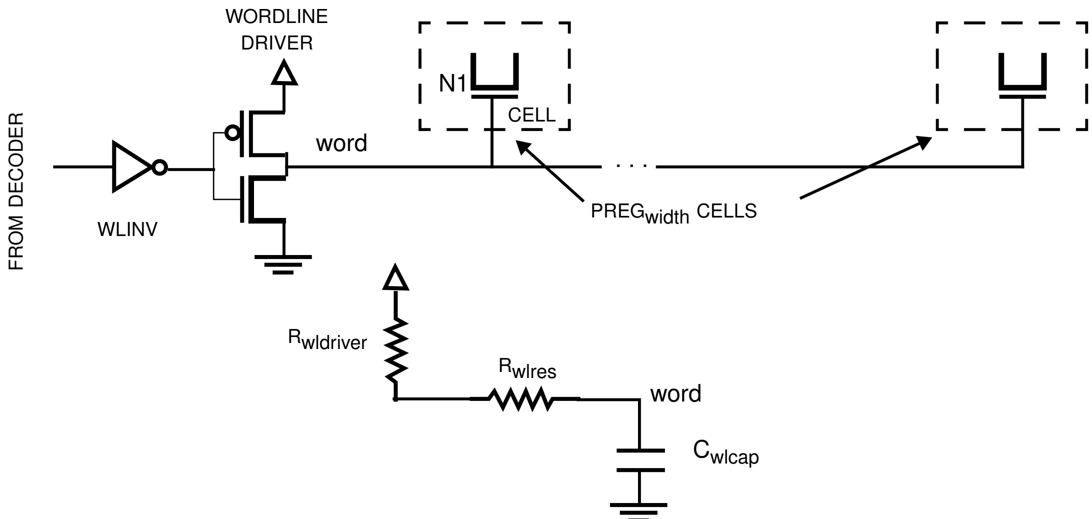


Figure 2-7. Wordline structure and equivalent circuit.

C_{eq} consists of three components: the diffusion capacitance of the NAND gate, the gate capacitance of the NOR gate, and the metal capacitance of the predecode wire. Hence,

$$C_{eq} = C_{diffcapnand} + C_{gatecapnor} + PredeclineLength \times C_{metal}$$

Substituting the above equations into the overall decoder delay and simplifying, we get

$$T_{decode} = c_0 + c_1 \times IW + c_2 \times IW^2$$

where c_0, c_1, and c_2 are constants. The quadratic component results from the intrinsic RC delay of the predecode lines connecting the NAND gates to the NOR gates. We found that, at least for the design space and technologies we explored, the quadratic component is very small relative to the other components. Hence, the delay of the decoder is linearly dependent on the issue width. Typical values for the constants are listed in Table B.1 in Appendix B.

Wordline delay

The wordline delay is defined as the time taken to turn on all the access transistors (denoted by N1 in Figure 2.7) connected to the wordline after the logical register designator has been decoded. The wordline delay is the sum of the delay of the inverter WLINV and the delay of the wordline driver. Hence,

$$T_{wordline} = T_{wlinv} + T_{wldriver}$$

From the equivalent circuit of the wordline driver shown in the figure, the wordline driver can be written as

$$T_{wldriver} = c_0 \times (R_{wldriver} + R_{wlres}) \times C_{wlcap}$$

where $R_{wldriver}$ is the effective resistance of the pull-up (p-transistor) of the driver, R_{wlres} is the resistance of the wordline, and C_{wlcap} is the amount of capacitance on the wordline. The total capacitance on the wordline consists of two components: the gate capacitance of the access transistors and the metal capacitance of the wordline wire. The resistance of the wordline is determined by the length of the wordline. Symbolically,

$$WordlineLength = (cellwidth + 6 \times IW \times bitline_{spacing} + sreg_{width}) \times PREG_{width}$$

$$C_{wlcap} = PREG_{width} \times C_{gatecapN1} + WordlineLength \times C_{metal}$$

$$R_{wlres} = 0.5 \times WordlineLength \times R_{metal}$$

where $PREG_{width}$ is the number of bits in the physical register designator, $C_{gatecapN1}$ is the gate capacitance of the access transistor N1 in each cell, *cellwidth* is the width of a single RAM cell excluding the bitlines, $bitline_{spacing}$ is the spacing between bitlines, and $sreg_{width}$ is the width of a single bit of the shift register in each cell.

Factoring the above equations into the wordline delay equation and simplifying we get

$$T_{wordline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

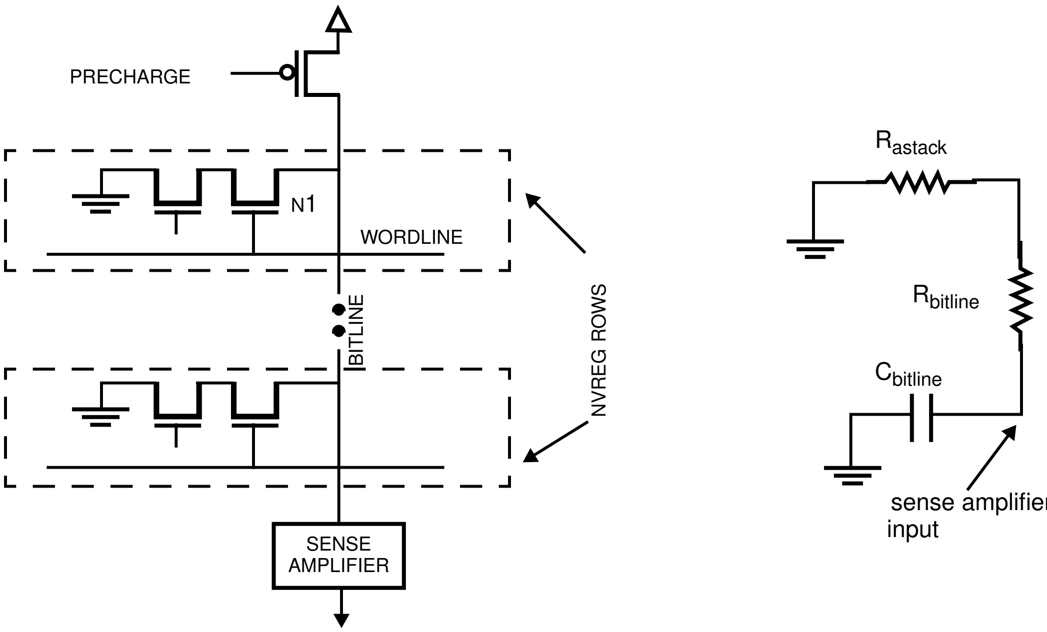


Figure 2-8. Bitline structure and equivalent circuit. We used Wada's sense amplifier [WRP92].

where c_0, c_1, and c_2 are constants. Again, the quadratic component results from the intrinsic RC delay of the wordline wire and we found that the quadratic component is very small relative to the other components. Hence, the overall wordline delay is linearly dependent on the issue width. Typical values for the constants are listed in Table B.2 in Appendix B.

Bitline delay

The bitline delay is defined as the time between the wordline going high (turning on the access transistor N1 shown in Figure 2-8) and the output of the sense amplifier going high/low. From the figure this is the sum of the time it takes for one access stack to discharge the bitline and the time it takes for a sense amplifier to detect the discharge. Hence,

$$T_{bitline} = T_{bitdischarge} + T_{senseamp}$$

From the equivalent circuit shown in the figure, the time taken to discharge the bitlines is determined by the following equations.

$$BitlineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NVREG$$

$$R_{bl} = 0.5 \times BitlineLength \times R_{metal}$$

$$C_{bl} = NVREG \times C_{diffcap} + BitlineLength \times C_{metal}$$

$$T_{bitdischarge} = c_0 \times (R_{astack} + R_{bl}) \times C_{bl}$$

where R_{astack} is the effective resistance of the access stack (two pass transistors in series), R_{bl} is the resistance of the bitline, C_{bl} is the capacitance on the bitline, *NPREG* is the number of physical registers, $C_{diffcap}$ is the diffusion capacitance of the access stack that connects to the bitline, *cellheight* is the height of a single RAM cell excluding the wordlines, and $wordline_{spacing}$ is the spacing between wordlines.

Factoring the above equations into the overall delay equation and simplifying we get

$$T_{bitline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

where c_0, c_1, and c_2 are constants. Again, we found that the quadratic component is very small relative to the other components. Hence, the overall bitline delay is linearly dependent on the issue width.

Overall delay

From the above analysis, the overall delay of the register rename logic can be summarized by the following equation

$$Delay = c_0 + c_1 \times IW + c_2 \times IW^2$$

where c_0, c_1, and c_2 are constants. However, the quadratic component is relatively small and hence, the rename delay is a linear function of the issue width for the design space we explored. Typical values for the constants are listed in Table B.3 in Appendix B.

2.4.1.3 Spice Results

Figure 2-9 shows how the delay of the rename logic varies with the issue width i.e. the number of instructions being renamed every cycle for the three technologies. The graph also shows the breakdown of the delay into the components discussed in the previous section. Detailed results for various configurations and technologies are shown in tabular form in Appendix A.

A number of observations can be made from the graph. The total delay increases linearly with issue width for the technologies. This is in conformance with the analysis in the previous section. All the components show a linear increase with issue width. The increase in the bitline delay is larger than the increase in the wordline delay because the bitlines are longer than the wordlines in our design. The bitline length is proportional to the number of logical registers (32 in most cases) whereas the wordline length is proportional to the width of the physical register designator (less than 8 for the design space we explored)

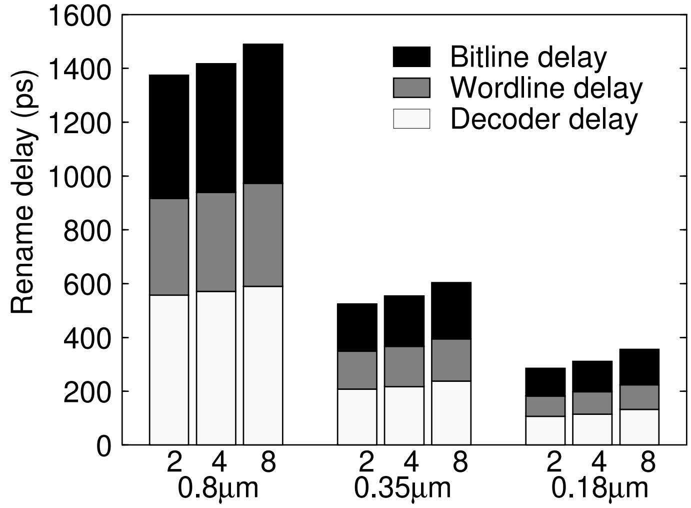


Figure 2-9. Rename delay versus issue width. This graph shows the breakup of rename delay for issue widths of 2, 4, and 8 for the three technologies.

Another important observation that can be made from the graph is that the relative increase in wordline delay, bitline delay, and hence, total delay with issue width only worsens as the feature size is reduced. For example, as the issue width is increased from 2 to 8, the percentage increase in bitline delay shoots up from 37% to 53% as the feature size is reduced from 0.8µm to 0.18µm. This occurs because logic delays in the various components are reduced in proportion to the feature size while the presence of wire delays in the wordline and bitline components cause the wordline and bitline components to fall at a slower rate. In other words, wire delays in the wordline and bitline structures will become increasingly important as feature sizes are reduced.

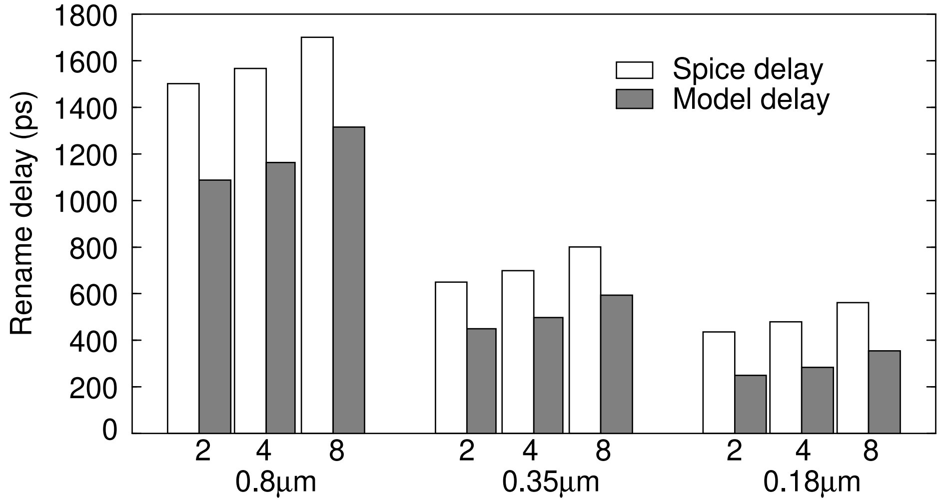


Figure 2-10. Model delay results for rename logic. This graph shows how the model delay results compare to the Spice results for register rename logic.

2.4.1.4 Model Results

Figure 2-10 shows how the delays computed by the model using the constants listed in Appendix B compare to the Spice results presented earlier. The delays computed by the analytical models, both for rename logic and for other structures to be presented later, are not always close to the Spice delays. The differences arise due to a number of reasons. First, the simple RC analysis makes a number of approximations and simplifications that cause deviation from the Spice result. Second, the simple delay equations used here do not take into account the slopes of input signals. Third, we could not find reliable delay models for quantifying the delay of dynamic gates. Since it is beyond the scope of the thesis, no attempt was made to develop advanced delay models tailored for this study. However, the analytical models for the different structures help establish dependence relationships and identify components that will become increasingly important in future.

2.4.2 Window Wakeup Logic

The wakeup logic is responsible for updating source dependences of instructions in the issue window waiting for their source operands to become available. Figure 2-11 illustrates the wakeup logic. Every time a result is produced, the tag associated with the result

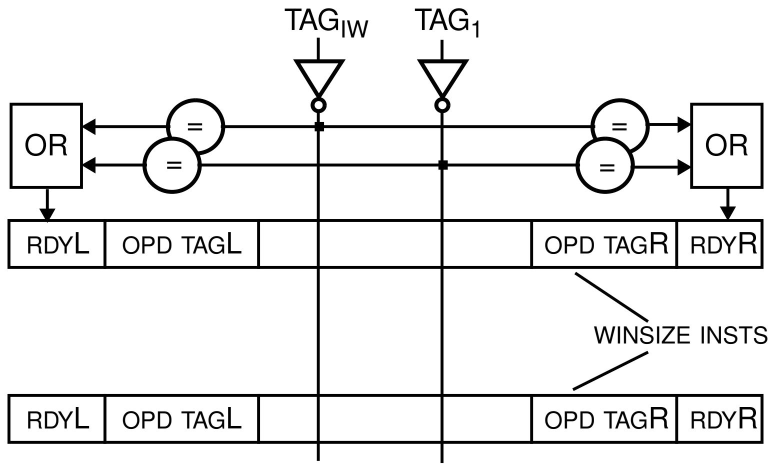


Figure 2-11. Window wakeup logic.

is broadcast to all the instructions in the issue window. Each instruction then compares the tag with the tags of its source operands. If there is a match, the operand is marked available by setting the *rdyL* or *rdyR* flag. Once all the operands of an instruction become available (both *rdyL* and *rdyR* are set), the instruction is ready to execute and the *rdy* flag is set to indicate this. The issue window is a CAM (content-addressable memory [WE93]) array holding one instruction per entry. Buffers, shown at the top of the figure, are used to drive the result tags tag_1 to tag_{IW} where IW is the issue width. Each entry of the CAM has ($2 \times IW$) comparators to compare each of the result tags against the two operand tags of the entry. The OR logic combines the comparator outputs and sets the *rdyL*/*rdyR* flags.

2.4.2.1 Structure

Figure 2-12 shows a single cell of the CAM array. The cell shown in detail compares a single bit of the operand tag with the corresponding bit of the result tag. The operand tag bit is stored in the RAM cell. The corresponding bit of the result tag is driven on the tag lines. The match line is precharged high. If there is a mismatch between the operand tag bit and the result tag bit, the match line is pulled low by one of the pull-down stacks. For example, if $tag = 0$, and $data = 1$, then the pull-down stack on the left is turned on and it pulls the match line low. The pull-down stacks constitute the comparators shown in Figure 2-12. The matchline extends across all the bits of the tag i.e. a mismatch in any of

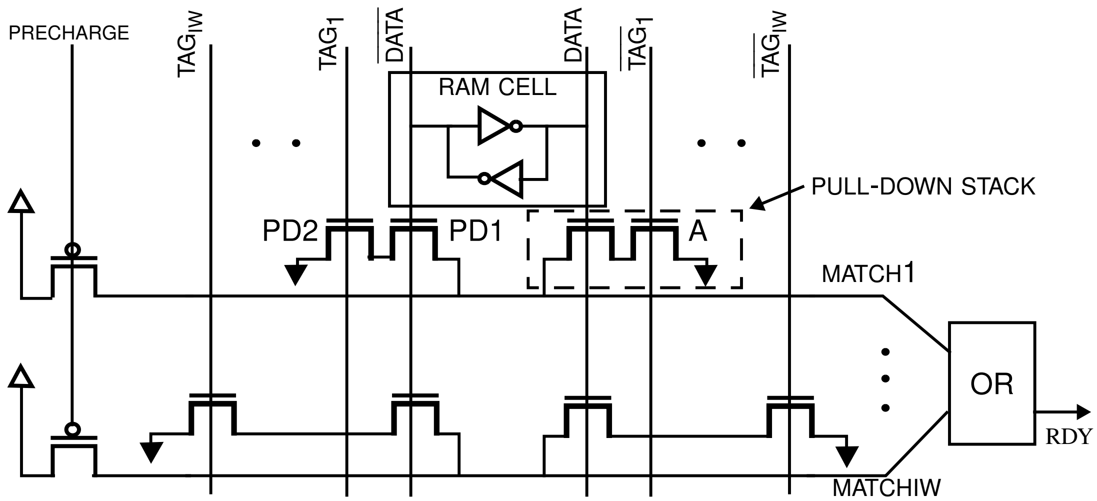


Figure 2-12. CAM cell in wakeup logic.

the bit positions will pull it low. In other words, the matchline remains high only if the result tag matches the operand tag. The above operation is repeated for each of the result tags by having multiple tag and matchlines as shown in the figure. Finally, all the match signals are ORed to produce the ready signal.

There are two observations that can be drawn from the figure. First, there are as many matchlines as the issue width. Hence, increasing issue width increases the height of each CAM row. Second, increasing issue width also increases the number of inputs to the OR block.

2.4.2.2 Delay Analysis

Because the match lines are precharged high, the default value of the ready signal is high. Hence, the delay of the critical path is the time it takes for a mismatch in a single bit position to pull the ready signal low. The delay consists of three components: the time taken by the buffers to drive the tag bits, the time taken for the pull-down stack corresponding to the bit position with the mismatch to pull the match line low, and the time taken to OR the individual match signals. Symbolically,

$$Delay = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

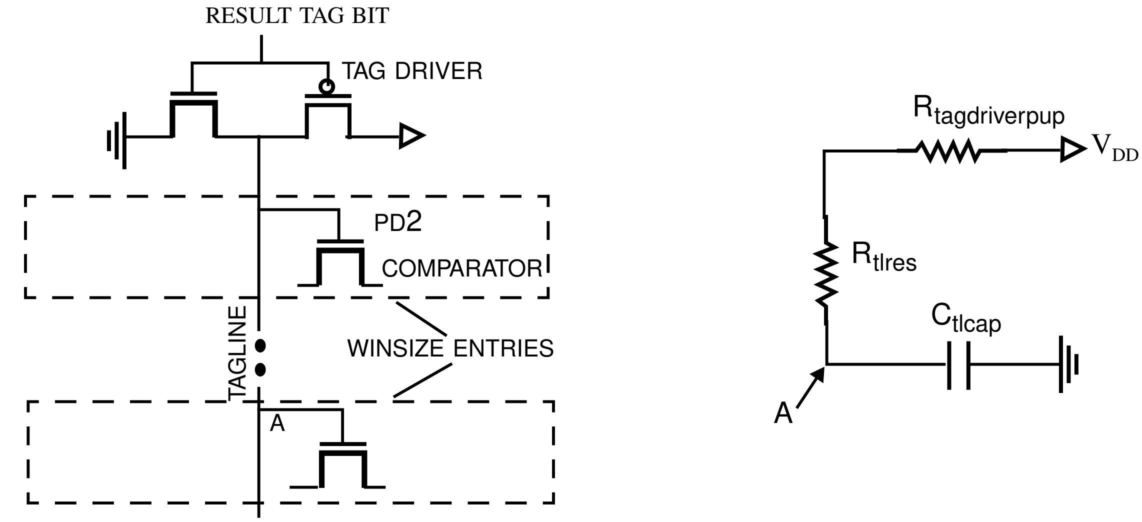


Figure 2-13. Tag drive structure.

Each of the components is analyzed next.

Tag Drive Time

The tag drive circuit is shown in Figure 2-13. The time taken to drive the tags depends on the length of the tag lines. The length of the tag lines is given by

$$TaglineLength = (camheight + IW \times matchline_{spacing}) \times WINSIZE$$

where *camheight* is the height of a single CAM cell excluding the matchlines, and $matchline_{spacing}$ is the spacing between matchlines[1].

From the equivalent circuit shown in the figure, the time taken to drive the tags is given by

$$T_{tagdrive} = c_0 \times (R_{tagdriverpup} + R_{tlres}) \times C_{tlcap}$$

where $R_{tagdriverpup}$ is the resistance of the pull-up of the tag driver, R_{tlres} is the metal resistance of the tag line, and C_{tlcap} is the total capacitance on the tag line. R_{tlres} is determined

1. To be precise $matchline_{spacing}$ is the height of a matchline and the associated pull-down stacks.

by the length of the tag lines. C_{tlcap} consists of three components: the metal capacitance determined by the length of the tag line, the gate capacitances of the comparators, and the diffusion capacitance of the tag driver. Symbolically,

$$R_{tlres} = 0.5 \times TaglineLength \times R_{metal}$$

$$C_{tlcap} = TaglineLength \times C_{metal} + C_{gatecapcomp} \times WINSIZE + C_{diffcap}$$

where $C_{gatecapcomp}$ is the gate capacitance of the pass transistor PD2 (shown in Figure 2-13) in the comparator's pull-down stack and $C_{diffcap}$ is the diffusion capacitance of the tag driver.

Substituting the above equations into the overall delay equation and simplifying we get

$$T_{tagdrive} = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c_5 \times IW^2) \times WINSIZE^2$$

The above equation shows that the tag drive time increases with window size and issue width. For a given issue width, the total delay is a quadratic function of the window size. The weighting factor for the quadratic term is a function of the issue width. We found that the weighting factor becomes significant for issue widths beyond 2. For a given window size, the tag drive time is also a quadratic function of issue width. We found that for current technologies (0.35µm and longer) the quadratic component is relatively small and the tag drive time is largely a linear function of issue width. However, as the feature size is reduced to 0.18µm the quadratic component also increases in significance. The quadratic component results from the intrinsic RC delay of the tag lines. The constants in the equation are listed in Table B.5 in Appendix B.

In reality, both issue width and window size will be simultaneously increased because a larger window is required for finding more independent instructions. Hence, we believe that the tag drive time can become significant in future designs with wider issue widths, bigger windows, and smaller feature sizes.

Tag Match time

The tag match time is the time taken for one of the pull-down stacks to pull the matchline low. From the equivalent circuit shown in Figure 2-14,

$$T_{tagmatch} = c_0 \times (R_{pdstack} + R_{mlres}) \times C_{mlcap}$$

where $R_{pdstack}$ is the effective resistance of the pull-down stack, R_{mlres} is the metal resistance of the matchline, and C_{mlcap} is the total capacitance on the match line. R_{mlres} can be computed using

$$MatchlineLength = (camwidth + IW \times tagline_{spacing}) \times PREG_{width}$$

$$R_{mlres} = 0.5 \times MatchlineLength \times R_{metal}$$

where *MatchlineLength* is the length of the matchlines, *camwidth* is the width of the CAM cell excluding the tag lines, $tagline_{spacing}$ is the spacing between tag lines.

C_{mlcap} consists of three components: the diffusion capacitance of all the pull-down stacks connected to the matchline, the metal capacitance of the matchline, and the gate capacitance of the inverter at the end of the matchline. Hence,

$$C_{mlcap} = 2 \times PREG_{width} \times C_{diffcap} + MatchlineLength \times C_{metal} + C_{gatecap}$$

where $PREG_{width}$ is the width of the physical register designators, $C_{diffcap}$ is the diffusion capacitance of the pass transistor (marked as PD1 in Figure 2-14) in the pull-down stacks

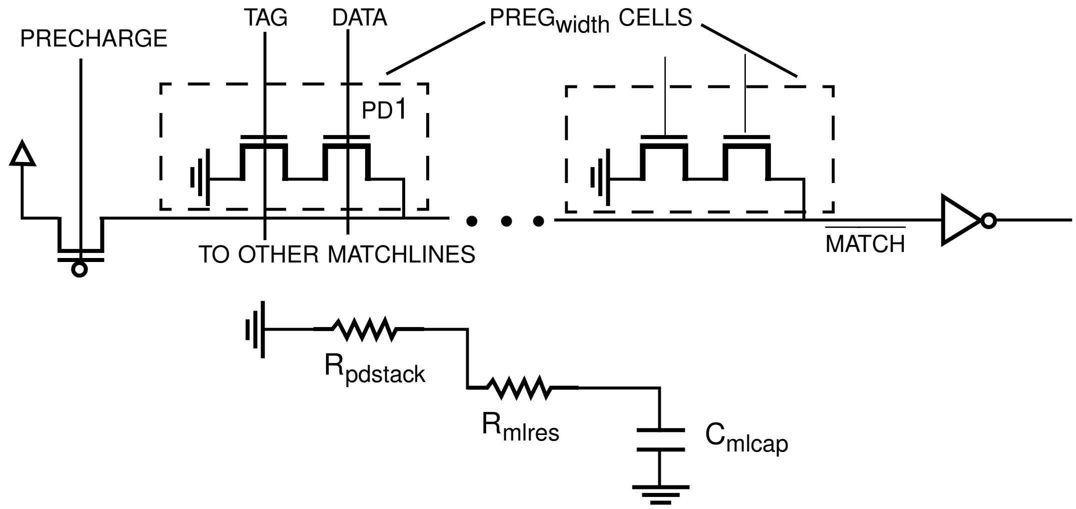


Figure 2-14. Tag match structure.

that is connected to the matchline, and $C_{gatecap}$ is the gate capacitance of the inverter at the end of the match line.

Substituting the equations for R_{mlres} and C_{mlcap} into the overall delay equation and simplifying we get

$$T_{tagmatch} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Again, we found that the quadratic component is relatively small and hence, the tag match time is a linear function of issue width. The constants are listed in Table B.6 in Appendix B.

A drawback of our model for the tag match time is that it does not model the dependence of the match time on the slope of the tag line signal i.e. the tag drive delay. Our results, presented in the next section, show that, as a result of this dependence, the tag match time is also a function of the window size. In other words, a larger window will result in slower fanning out of the result tags to the comparators in the window entries, thus increasing the compare time.

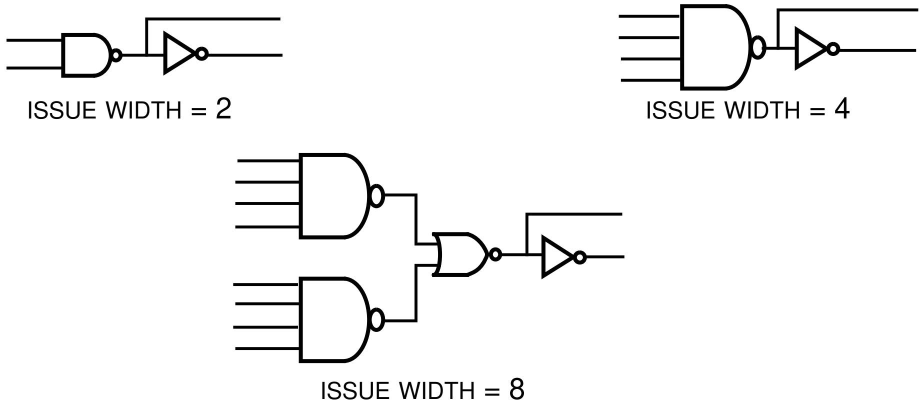


Figure 2-15. Logic for ORing individual match signals.

Match OR time

This is the time taken to OR the individual matchlines to produce the ready signal. Because the number of matchlines is the same as the issue width, the magnitude of this delay term is a direct function of issue width. Figure 2-15 shows the OR logic for result widths of 2, 4, and 8. For an issue width of 8, we use two 4-input NAND stacks followed by a NOR gate because this is faster than using an 8-input NAND gate. Because the rise delay of a gate is a linear function of the of the fan-in [WE93,Rab96] we can write the delay as

$$T_{matchOR} = c_0 + c_1 \times IW$$

where the constants are as shown in Table B.7 in Appendix B.

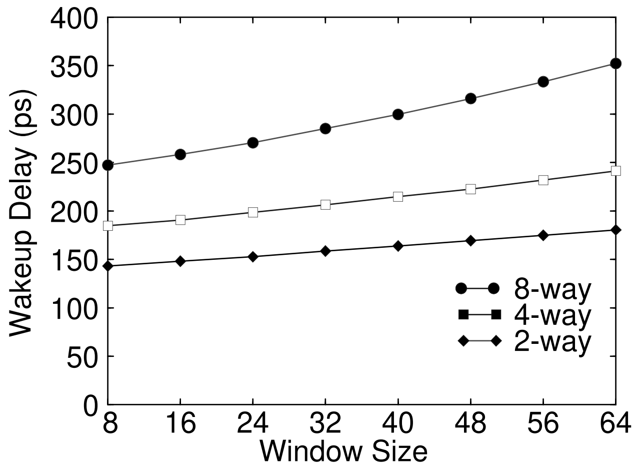


Figure 2-16. Wakeup logic delay versus window size. This graph shows how the delay of the window wakeup logic varies with window size and issue width for 0.18µm technology.

Overall delay

The overall delay of the wakeup logic can be summarized by the following equation:

$$
\begin{aligned}
Delay &= (c_0 + c_1 \times IW + c_2 \times IW^2) \\
&+ (c_3 + c_4 \times IW) \times WINSIZE \\
&+ (c_5 + c_6 \times IW + c_7 \times IW^2) \times WINSIZE^2
\end{aligned}
$$

where the constants are as tabulated in Table B.8 in Appendix B.

2.4.2.3 Spice Results

The graph in Figure 2-16 shows how the delay of the wakeup logic varies with window size and issue width for 0.18µm technology. As expected, the delay increases as window size and issue width are increased. The quadratic dependence of the total delay on the window size results from the quadratic increase in tag drive time as discussed in the previous section. This effect is clearly visible for issue width of 8 and is less significant for smaller issue widths. We found similar curves for 0.8µm and 0.35µm technologies. The quadratic dependence of delay on window size was more prominent in the curves for 0.18µm technology than for the other two technologies

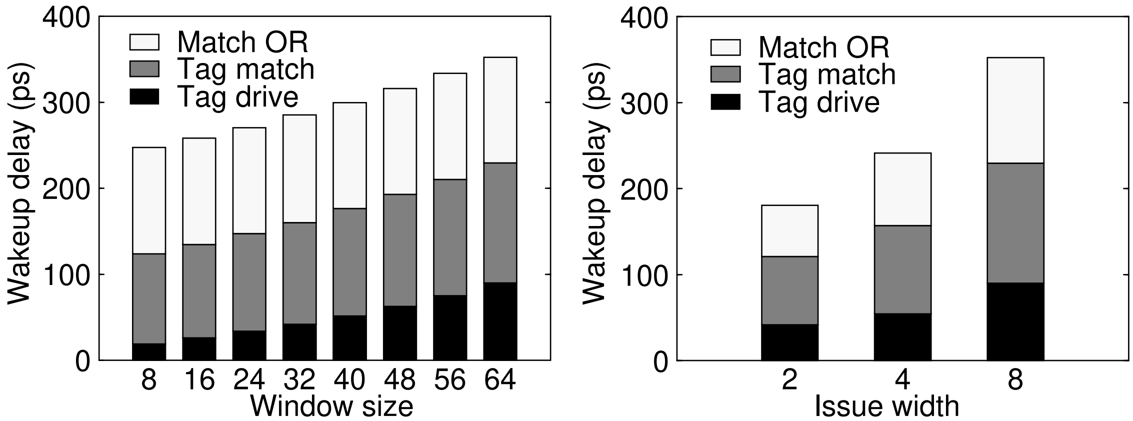


Figure 2-17. Wakeup logic delay. The graph on the left shows how wakeup delay varies with window size for a 8-way machine. The graph on the right shows how wakeup delay varies with issue width for a 64-entry window. Both graphs are for 0.18μm technology.

Also, issue width has a greater impact on the delay than window size because increasing issue width increases all the three components of the delay. On the other hand, increasing window size only lengthens the tag drive time and to a small extent the tag match time. Overall, the results show that the delay increases by almost 34% going from 2-way to 4-way and by 46% going from 4-way to 8-way for a window size of 64 instructions. In reality, the increase in delay is going to be even worse because in order to sustain a wider issue width, a larger window is required to find independent instructions. We found similar curves for 0.8μm and 0.35μm technologies. Detailed results for various configurations and technologies are shown in tabular form in Appendix A.

The bar graph on the left in Figure 2-17 shows the detailed breakdown of the total delay for various window sizes for a 8-way processor in 0.18μm technology. The tag drive time increases rapidly with window size. For example, the tag drive time and the tag match time increase by factors of 4.78 and 1.33 respectively when the window size is increased from 8 to 64. The increase in tag drive time is higher than that of tag match time because the tag drive time is a quadratic function of the window size. The increase in tag match time with the window size is not taken into account by our simple model given above because the model does not take into consideration the slope of the input signals (determined in this case by the tag drive delay). Also, as shown by the graph, the time taken to OR the match

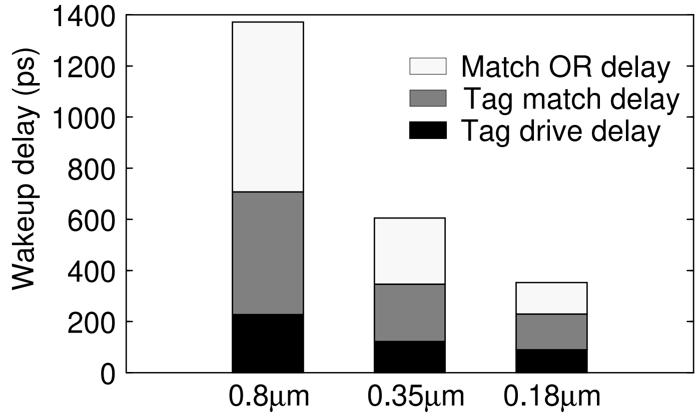


Figure 2-18. Wakeup delay versus feature size. This graph shows how the wakeup delay for a 8-way machine with a 64-entry window varies with feature size.

signals depends only on the issue width and is independent of the window size The graph on the right in Figure 2-17 shows how the delay of a 64-entry window in 0.18μm technology varies with issue width. As shown by the delay analysis, all the components increase with issue width.

Figure 2-18 shows the effect of reducing feature sizes on the various components of the wakeup delay for an 8-way, 64-entry window processor. The tag drive and tag match delays do not scale as well as the match OR delay. This is expected because tag drive and tag match delays include wire delays whereas the match OR delay only consists of logic delays. Quantitatively, the fraction of the total delay contributed by tag drive and tag match delay increases from 52% to 65% as the feature size is reduced from 0.8μm to 0.18μm. This shows that the performance of the broadcast operation will become more critical in future technologies.

In the above simulation results the window size was limited to a maximum of 64 instructions because we found that for larger windows the intrinsic RC delay of the tag lines increases significantly. As discussed previously, the intrinsic RC delay is proportional to the square of the window size. Therefore, for implementing larger windows banking should be used. Banking helps alleviate the intrinsic RC delay by reducing the length of the tag lines. For example, two-way banking will improve the intrinsic RC delay by a fac-

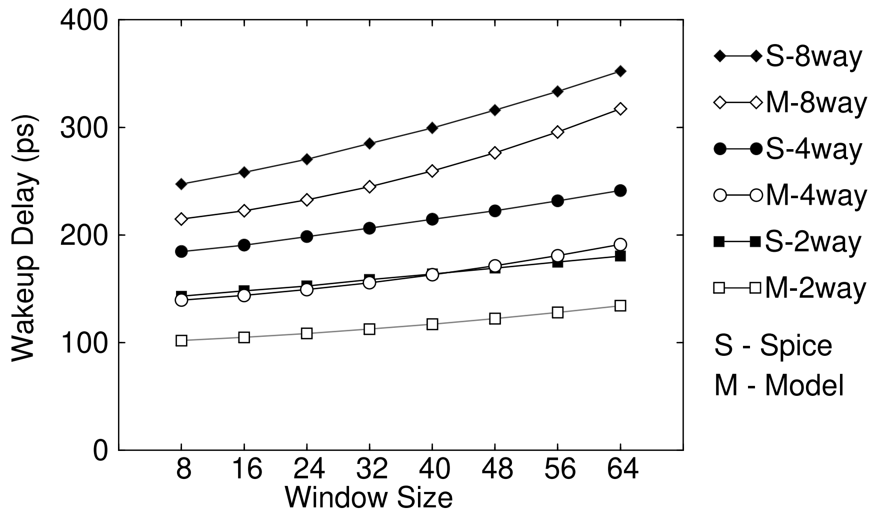


Figure 2-19. Model delay results for wakeup logic. This graph shows how the model delay results compare to the Spice results for 0.18μm technology.

tor of four. At the same time it must be pointed out that banking will introduce some extra delay due to extra inverter stages and the parasitics introduced by the extension to the tag lines.

2.4.2.4 Model Results

Figure 2-19 shows how the model results, computed using the constants in Appendix B, compare to the Spice results. From the graph we can see that the model is successful in tracking the dependence on issue width and window size.

2.4.3 Window Selection Logic

Selection logic is responsible for selecting instructions for execution from the pool of ready instructions in the issue window. Some form of selection logic is required for two reasons. First, the number of ready instructions in the issue window can be greater than the number of functional units. For example, for a machine with a 32-entry issue window there could be as many as 32 ready instructions. Second, some instructions can be executed only on a subset of the functional units. For example, if there is only one integer multiplier, all multiply instructions will have to be steered to that functional unit.

The inputs to the selection logic are the request signals, termed *REQ*, one per instruction in the issue window. The request signal of an instruction is raised when all the operands of the instruction become available. As discussed in the previous section, the wakeup logic is responsible for raising the *REQ* signals. The outputs of the selection logic are the grant signals, termed *GRANT*, one per request signal. On receipt of the *GRANT* signal, the associated instruction is issued to the functional unit and the corresponding window entry is freed for later use. A selection policy is used to decide which of the requesting instructions is granted the functional unit. We use a selection policy that is based on the location of the instruction in the window. The HP PA-8000 [Kum96] uses a similar selection policy. We chose this policy because it allows a simpler, and hence faster, implementation compared to other more sophisticated policies like “oldest ready first”.

2.4.3.1 Structure

The assumed structure of the selection logic is shown in Figure 2-20. The selection logic is used to select a single instruction for execution on a functional unit. The modifications to this scheme for handling multiple functional units is discussed later. The selection logic consists of a tree of arbiters. Each arbiter cell functions as follows. If the enable input is high, then the grant signal corresponding to the highest priority, active input is raised. For example, if *enable* = 1, *req0* = 0, *req1* = 1, *req2* = 0, and *req3* = 1, then *grant1* will be raised assuming priority reduces as we go from input *req0* to input *req3*. If the *enable* input is low, all the *grant* signals are set to low. In all cases, at most one of the *grant* signals is high. The *anyreq* output signal is raised if any of the input *req* signals is high

The overall selection logic works in two phases. In the first phase, the *request* signals are propagated up the tree. Each cell raises the *anyreq* signal if any of its input *request* signals is high. This in turns raises the input *request* signal of its parent arbiter cell. Hence, at the root cell one or more of the input *request* signals will be high if there are one or more instructions that are ready. The root cell then grants the functional unit to one of its children by raising one of its *grant* outputs. This initiates the second phase. In this phase, the

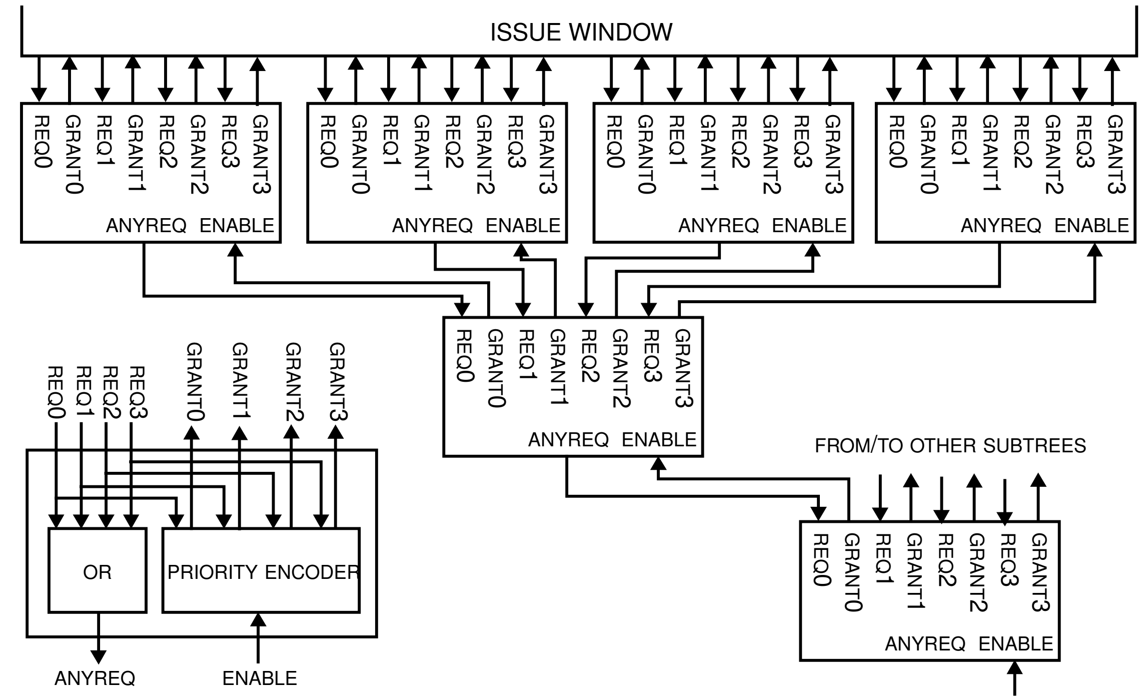


Figure 2-20. Selection logic. This figure shows the arbiter tree of the selection logic and a single arbiter cell in detail.

grant signal is propagated down the tree to the instruction that is selected. At each level, the *grant* signal is propagated down the subtree that contains the selected instruction. The *enable* signal to the root cell is high whenever the functional unit is ready to execute an instruction. For example, for single-cycle ALUs, the *enable* signal will be permanently tied to high.

.The selection policy implemented by our assumed structure is static and is strictly based on location of the instruction in the window. The leftmost entries in the window have the highest priority. The oldest ready first policy can be implemented using our scheme by compacting the issue window to the left every time instructions are issued and by inserting new instructions at the right end. This ensures that instructions that occur earlier in program order occupy the leftmost entries in the window and hence have higher priority than later instructions. However, it is possible that the complexity resulting from compaction

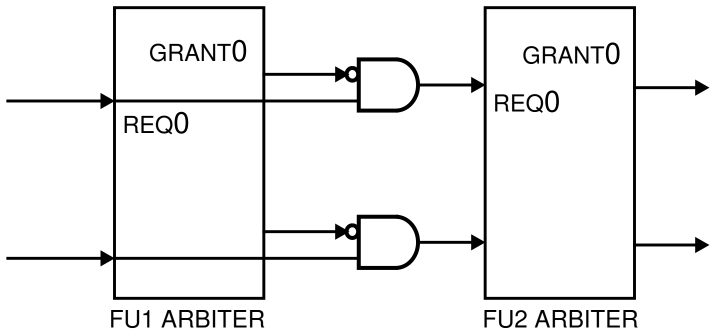


Figure 2-21. Handling multiple functional units.

could degrade performance. We did not analyze the complexity of compacting in this study.

Handling Multiple Functional Units

If there are multiple functional units of the same type, then selection logic (shown in Figure 2-21) comprises a number of blocks of the type studied in the previous section, stacked in series. The *request* signals to each block are derived from the requests to the previous block by masking the request that was granted the previous resource.

An alternative to the above scheme is to extend the arbiter cells so that the request and grant signals encode the number of resources being requested and granted respectively. However, we believe that this could considerably slow down the arbiter cells and hence could perform worse than the stacked design. The stacked design might not be a feasible alternative beyond two functional units because the resulting delay can be significant. An alternative option is to statically partition the window entries among the functional units. For example, in the MIPS R10000 [Yea96], the window is partitioned into three sets called the integer queue, floating-point queue, and the address queue. Only instructions in the integer queue are monitored for execution on the two integer functional units.

2.4.3.2 Delay Analysis

The delay of the selection logic is the time it takes to generate the grant signal after the request signal has been raised. This is equal to the sum of two terms: the time taken for the request signal to propagate to the root of the tree and the time taken for the grant signal to propagate from the root to the selected instruction. Symbolically,

$$Delay = (L-1) \times T_{reqpropd} + T_{root} + (L-1) \times T_{grantpropd}$$

where $L = log_4(WINSIZE)$ is the height of the selection tree, $T_{reqpropd}$ is the time taken for the request signal to propagate through an arbiter cell, T_{root} is the delay of the grant output at the root cell, and $T_{grantpropd}$ is the time taken for the grant signal to propagate through an arbiter cell. Hence, the overall delay can be written as

$$Delay = c_0 + c_1 \times \log_4(WINSIZE)$$

where c_0 and c_1 are constants as listed in Table B.9 in Appendix B. The base of the logarithmic term is determined by the number of inputs to the arbiter. We found the optimal number of arbiter inputs to be four in our case. The associated trade-offs are discussed later.

From the above equations we can see that the delay of the selection logic is proportional to the height of the tree and the delay of the arbiter cells. The delay has a logarithmic relationship with the window size. Increasing issue width can also increase the selection delay if a stacked scheme is used to handle multiple functional units. For the rest of the discussion, we will assume that a single functional unit is being scheduled and hence no stacking is used. The delay for a stacked design can be easily computed by multiplying our delay results by the stacking depth. One way to improve the delay of the selection logic is to increase the radix of the selection tree. However, as we will see shortly, this increases the delay of a single arbiter cell and could make the overall delay worse.

Arbiter Logic

The circuit for generating the *anyreq* signal is shown in Figure 2-22. The *anyreq* signal is raised if one or more of the input request signals is active. The circuit, implementing the OR function, consists of a dynamic NOR gate followed by an inverter. The dynamic gate was chosen instead of a static OR gate for speed reasons. The circuit operates as follows. The $\overline{anyreq}$ node is precharged high. When one or more of the input request signals go high, the corresponding pull-downs pull the $\overline{anyreq}$ node low. The inverter in turn raises the *anyreq* signal high. The value of $T_{reqpropd}$ in the delay equation is the delay of the OR circuit.

The priority encoder in the arbiter cell is responsible for generating the grant signals. The logic equations for the grant signals are:

$$grant0 = req0 \cap enable$$

$$grant1 = \overline{req0} \cap req1 \cap enable$$

$$grant2 = \overline{req0} \cap \overline{req1} \cap req2 \cap enable$$

$$grant3 = \overline{req0} \cap \overline{req1} \cap \overline{req2} \cap req3 \cap enable$$

For example, *grant2* is high only if the cell is enabled, the input requests *req0* and *req1* are low, and *req2* is high. Because the request signals at each cell, except for the root, are available well in advance of the enable signal we use a two-level implementation for evaluating the grant signals. As an example, the circuit for evaluating *grant1* is shown in Figure 2-22. The first stage evaluates the *grant1* signal (node grant1p) assuming the enable signal is high. In the second stage, the *grant1p* signal is ANDed with the enable to produce the *grant1* signal. This two-level decomposition was chosen because it removes

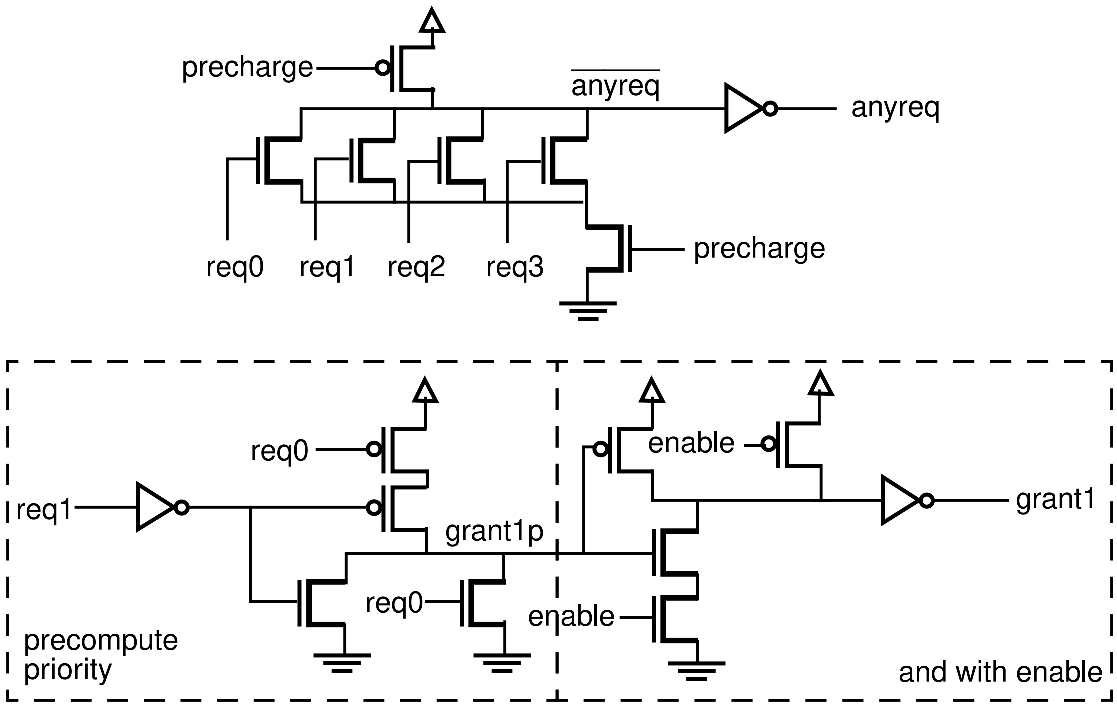


Figure 2-22. Arbiter Logic. The block on top shows the logic for the *anyreq* signal. The bottom block shows the logic for generating the *grant1* signal.

the logic for *grant1p* from the critical path. This optimization does not apply at the root cell because at the root cell the request signals arrive after the enable signal.

The policy used by the selection logic is embedded in the above equations for the *grant* outputs of the arbiter cell. For example, the design presented assumes static priority with *req0* having the highest priority. Implementing an alternative policy would require appropriate modifications to these equations. Again, the designer has to be careful while selecting a policy because a complex policy can increase the delay of the selection logic by slowing down individual arbiter cells.

Increasing the number of inputs to the arbiter cell slows down both the OR logic and the priority encoder logic. The OR logic slows down because the load capacitance contributed by the diffusion capacitance of the pull-downs increases linearly with the number of inputs. The priority logic slows down because the delay of the logic used to compute pri-

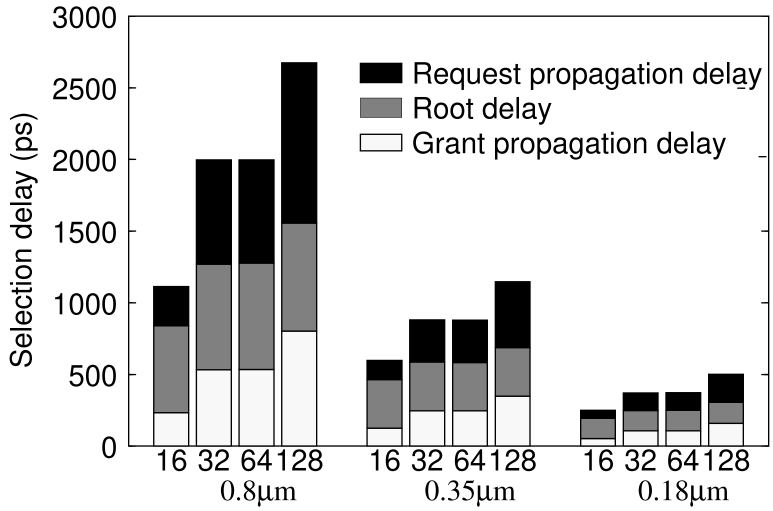


Figure 2-23. Selection delay versus window size. This graph shows how the selection delay varies with window size for the three different feature sizes. The selection policy used is based on the location of the instruction in the window.

ority increases due to higher fan-in. We found the optimal number of inputs to be four in our case. The selection logic in the MIP R10000, described in [V+96], is also based on four-input arbiter cells.

2.4.3.3 Spice Results

Figure 2-23 shows the delay of the selection logic for various window sizes in the three technologies assuming a single functional unit is being scheduled. The delay is broken down into the three components discussed earlier. From the graph we can see that for all three technologies, the delay increases logarithmically with window size. Also, the increase in delay is less than 100% when the window size is increased from 16 instructions to 32 instructions (or from 64 instructions to 128 instructions) because the middle term in the delay equation, the delay at the root cell, is independent of the window size. Detailed results are presented in tabular form in Appendix A.

The various components of the total delay scale well as the feature size is reduced. This is not surprising because all the delays are logic delays. It must be pointed out that the selection delays presented here are optimistic because we do not consider the wires in the circuit, especially if it is the case that the request signals originate from the CAM entries

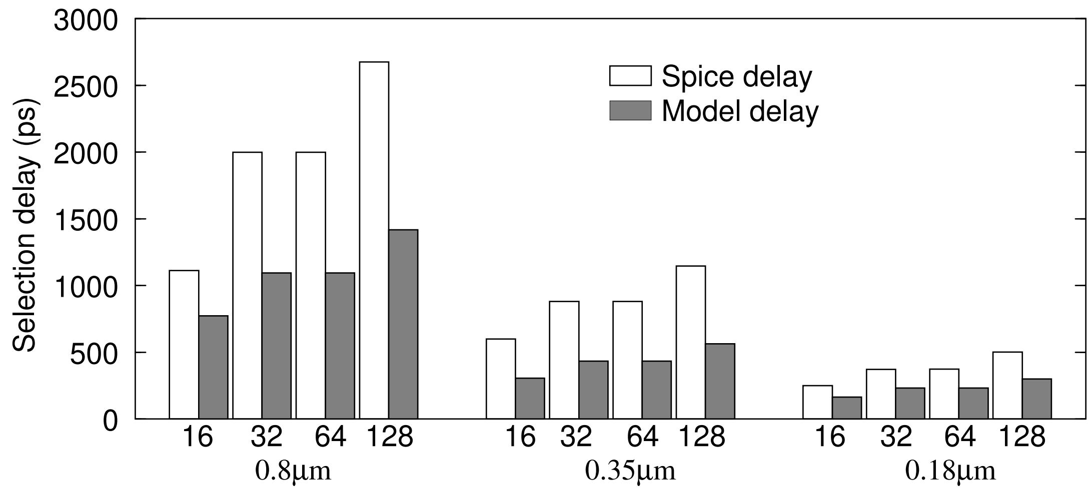


Figure 2-24. Model delay results for selection logic. This graph shows how the model delay results compare to the Spice results for selection logic.

in which the instructions reside. On the other hand, it might be possible to minimize the effect of these wire delays if the ready signals are stored in a smaller, more compact array.

2.4.3.4 Model Results

Figure 2-24 shows how the model delay results, computed using the constants listed in Appendix B, compare to the Spice results. The significant difference, especially for 0.8µm technology, results because our delay models are unable to accurately model dynamic logic.

2.4.4 Register file Logic

The register file provides low latency access to register operands. The access time of the register file depends on the number of registers in the file and the number of ports into the file. Assuming two read operands and one write operand per instruction, the number of read and write ports required for a machine with issue width *IW* is $2 \times IW$ and *IW* respectively[1]. The number of registers required increases with issue width in order to support a greater degree of speculative execution. A recent study [FJC96] shows that for significant

1. In most machine designs additional write ports are implemented for write-back of load data.

performance up to 80 physical registers are required for a 4-wide issue machine and up to 120 physical registers are required for an 8-wide issue machine.

2.4.4.1 Structure

The structure of the register file assumed for this study is similar to that of the map table shown in Figure 2-5 on page 29. The register file contents are stored in the cross-coupled inverters in the cells. Each row of cells stores the contents of a single register. Hence, the number of rows is determined by the number of registers in the file. The number of cells in each row is determined by the datapath width. We assume a 64-bit datapath for this study. A read operation starts with the register number (physical) being applied to the decoder. The decoder decodes the register number and raises one of the wordlines. This triggers bit line changes which are sensed by a sense amplifier and the appropriate output is generated. We use precharged, double-ended bitlines to improve the speed of read operations. Read ports are implemented using NAND stacks (two pass gates in series) instead of a single pass gate to prevent flipping of cell contents during a read operation, especially for configurations with a large number of read ports.

There are a few differences between the map table in the register rename logic and the register file. The shift register component of the map table is not present in the register file. In the case of the rename logic, the number of rows is determined by the number of logical registers in the instruction set architecture. The number of rows in the register file is determined by the number of physical registers. The width of each row in the map table is determined by the width of the physical register tags. In case of the register file, the width of each row is determined by the datapath width — 64 bits in most current designs.

2.4.4.2 Delay Analysis

The critical path for the register file logic is the time it takes for the contents of the register to be output after the register number is applied to the address decoder. The delay of the critical path consists of three components: the time taken to decode the register number, the time taken to drive the wordline, and the time taken by an access stack to pull the bitline low and for the sense amplifier to detect the change in the bitline and produce the corresponding output. Hence, the overall delay is given by,

$$T_{delay} = T_{decode} + T_{wordline} + T_{bitline}$$

Each of the components is analyzed next. The analysis presented here is similar to that presented for the rename logic. Hence, figures are omitted and the discussion is kept brief.

Decoder delay

We use the same predecoding scheme as used in the map table of the rename logic s shown in Figure 2-6 on page 31. The fan-in of the NAND and NOR gates is determined by the number of bits in the register number i.e. the number of physical registers. Table 2.2 shows the fan-in of the decoder gates for the various register file sizes simulated.

Number of physical registers	Fan-in of predecode gates	Fan-in of direct decode gates
32	2	3
64	2	3
128	3	3
256	4	2
512	4	3

Table 2.2: Fan-in of decoder gates.

The output of the NAND gates is connected to the input of the NOR gates by the predecode lines. The length of these lines is given by

$$PredeclineLength = 0.5 \times (cellheight + 3 \times IW \times wordline_{spacing}) \times NPREG$$

where *cellheight* is the height of a single cell excluding the wordlines, *IW* is the issue width, $wordline_{spacing}$ is the spacing between the wordlines, and *NPREG* is the number of physical registers. The factor 3 in the equation results from the assumption of 3-operand instructions (2 read operands and 1 write operand). With these assumptions, 3 ports (1 write port and 2 read ports) are required per cell for each instruction being renamed. Hence, for a *IW*-wide issue machine, a total of *(3 × IW)* wordlines are required for each cell. The factor 0.5 results from the assumption that the predecode NAND gates drive the predecode lines from the centre of the array. This optimization was necessary to minimize the RC effects of long predecode lines for large, highly ported configurations.

The decoder delay is the time it takes to decode the register number i.e. the time it takes for the output of the NOR gate to rise after the input to the NAND gate has been applied. Hence, the decoder delay can be written as

$$T_{decode} = T_{nand} + T_{nor}$$

where T_{nand} is the delay of the NAND gate and T_{nor} is the delay of the NOR gate. T_{nand} is given by the following equations,

$$T_{nand} = c_0 \times R_{eq} \times C_{eq}$$

$$R_{eq} = R_{nandpd} + 0.5 \times PredeclineLength \times R_{metal}$$

$$C_{eq} = C_{diffcapnand} + C_{gatecapnor} + PredeclineLength \times C_{metal}$$

where R_{nandpd} is the pull-down resistance of the NAND gate, $C_{diffcapnand}$ is the diffusion capacitance at the output of the NAND gates, $C_{gatecapnor}$ is the gate capacitance of the NOR gates.

Substituting the above equations into the overall decoder delay and simplifying, we get

$$T_{decoder} = c_0 + (c_1 + c_2 \times IW) \times NPREG + (c_3 + c_4 \times IW + c_5 \times IW^2) \times NPREG^2$$

The above equation shows that the decode time increases with the number of physical registers and the issue width. For a given issue width, the total delay is a quadratic function of register file size. The weighting factor for the quadratic term is a function of the issue width. For a given register file size, the decode time is also a quadratic function of issue width. The quadratic components in both cases result from the intrinsic RC delay of the predecode lines and are small relative to the other components. Typical values of the constants in the equation are listed in Table B.10 in Appendix B.

Wordline Delay

The wordline delay is defined as the time taken to turn on all the access transistors connected to the wordline after the register number has been decoded. The wordline delay is the sum of the fall delay of the *wlinv* inverter and the rise delay of the wordline driver. The delay of the wordline driver is given by the following equations

$$WordlineLength = (cellwidth + 6 \times IW \times bitline_{spacing}) \times DATA_{width}$$

$$R_{wl} = 0.5 \times WordlineLength \times R_{metal}$$

$$C_{wl} = DATA_{width} \times C_{gatecap} + WordlineLength \times C_{metal}$$

$$T_{wldriver} = c_0 \times (R_{wldriver} + R_{wl}) \times C_{wl}$$

where R_{wl} is the resistance of the wordline wire, C_{wl} is the capacitance on the wordline, $R_{wldriver}$ is the pull-up resistance of the wordline driver, and $C_{gatecap}$ is the gate capacitance of the access transistor.

Factoring the above equations into the wordline delay equation and simplifying we get

$$T_{wordline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

where c_0, c_1, and c_2 are constants listed in Table B.11 in Appendix B. Again, the quadratic component results from the intrinsic RC delay of the wordline wire and we found that this component is very small relative to other components. Hence, the overall wordline delay is linearly dependent on the issue width.

Bitline delay

The bitline delay is defined as the time between the wordline going high (turning on the access transistor N1) and the output of the sense amplifier going high/low. This is the sum of the time it takes for one access stack to discharge the bitline and the time it takes for a sense amplifier to detect the discharge. Hence,

$$T_{bitline} = T_{bitdischarge} + T_{senseamp}$$

The time taken to discharge the bitlines is determined by the following equations.

$$BitlineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NPREG$$

$$R_{bl} = 0.5 \times BitlineLength \times R_{metal}$$

$$C_{bl} = NPREG \times C_{diffcap} + BitlineLength \times C_{metal}$$

$$T_{bitdischarge} = c_0 \times (R_{astack} + R_{bl}) \times C_{bl}$$

where R_{astack} is the effective resistance of the access stack (two pass transistors in series), R_{bl} is the resistance of the bitline, C_{bl} is the capacitance on the bitline, *NPREG* is the number of physical registers, $C_{diffcap}$ is the diffusion capacitance of the access stack that connects to the bitline, *cellheight* is the height of a single RAM cell excluding the wordlines, and $wordline_{spacing}$ is the spacing between wordlines.

Factoring the above equations into the overall delay equation and simplifying we get

$$T_{bitline} = c_0 + (c_1 + c_2 \times IW) \times NPREG + (c_3 + c_4 \times IW + c_5 \times IW^2) \times NPREG^2$$

The bitline delay shows a similar dependence on issue width and register file size as the decoder delay. The quadratic components result from the intrinsic RC delay of the bitline wire. Again, we found that the quadratic component is very small relative to the other components. Typical values for the constants are listed in Table B.12 in Appendix B.

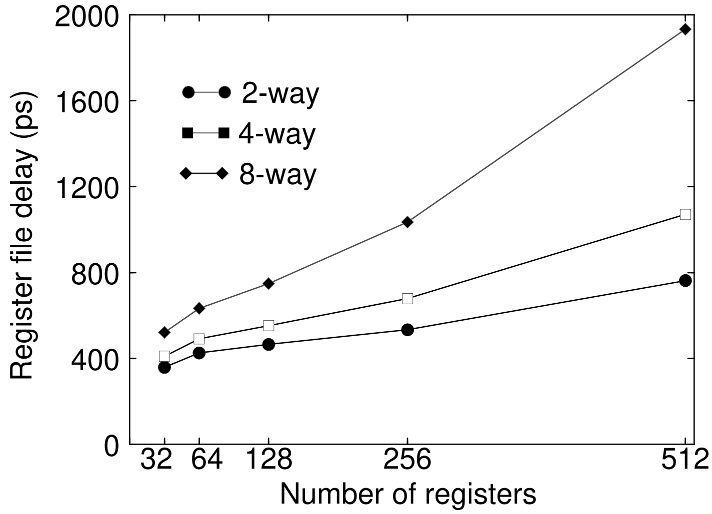


Figure 2-25. Register file logic delay. This graph shows how the delay of the register file implemented in 0.18µm varies with issue width and the number of registers

Overall delay

From the above analysis, the overall delay of the register file can be summarized by the following equation:

$$\begin{aligned} Delay &= (c_0 + c_1 \times IW + c_2 \times IW^2) \\ &+ (c_3 + c_4 \times IW) \times NPREG \\ &+ (c_5 + c_6 \times IW + c_7 \times IW^2) \times NPREG^2 \end{aligned}$$

where the constants are as tabulated in Table B.13 in Appendix B.

2.4.4.3 Spice Results

Figure 2-25 shows how the delay of the register file varies with the number of registers and the issue width for the case of 0.18µm technology. A number of observations can be made from the graph. First, the delay increases as issue width and the number of registers are increased. The graph also shows that the total delay is a linear function of the number of registers. The dependence on issue width is also linear except for larger configurations (512 registers or more) where the quadratic component start to show. These observations are in agreement with the analysis presented in the previous section. Issue width has a greater impact on the delay than the number of registers. This is expected because, as shown in the previous sections, increasing issue width increases all the three components of the total delay. For example, increasing the issue width from 4 to 8 increases the total delay by 28.9%, whereas increasing the number of registers from 64 to 128 for a 8-way machine only increases the delay by 18.1%. In practice, the increase in delay is going to be even worse because in order to sustain a wider issue width, more registers are required to support a larger number of speculative operations. We found similar curves for 0.8μm and 0.35μm technologies.

The graph in Figure 2-26 shows the breakdown of total delay into the components discussed in the previous section. The graphs are for the case of 0.18μm technology. Consider the graph on the left. As expected, the decoder delay and bitline delay increase with the number of registers. However, the decoder delay does not increase as smoothly as the bitline delay because the decoder structure (fan-in of the NAND and NOR gates) changes discretely with the number of registers as shown in Table 2.2. The wordline delay does not change with the number of registers because it is a function of the width of the registers (64 bits in our case) and the number of ports, both being constant for the graph. The bitline delay increases linearly with the number of registers because the capacitance on the bitlines increases linearly with the number of registers. The graph on the right shows how the breakdown varies with issue width for a 128-entry register file. In this case all three components increase with issue width. The decoder delay increases slightly with issue width even though its structure is determined by the number of registers because the predecode

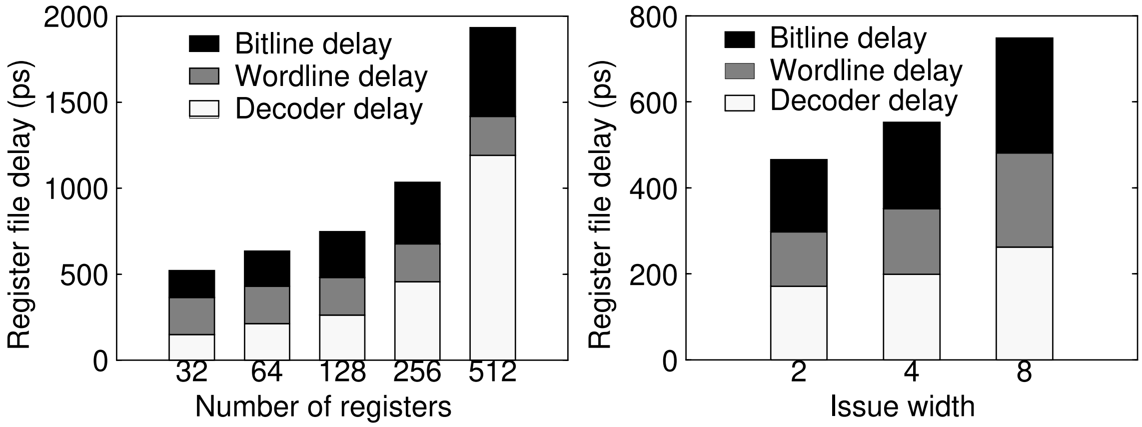


Figure 2-26. Breakup of register file delay. The graph on the left shows how the breakup varies with the number of registers for a 8-way machine in 0.18μm technology. The graph on the right shows how the breakup varies with issue width for a 128-entry register file in 0.18μm technology.

lines increase in length with issue width. The wordline and bitline components show a linear increase with issue width.

While the structure assumed for the above analysis is popular and has been used in most implementations, microprocessor vendors are beginning to explore alternatives that help reduce the delay of the register file. For example, the DEC 21264 [G$^+$97] uses two copies of the register file, each with half the number of read ports as the original file. Writes are sent to both copies. Each copy therefore has the same number of write ports at the original file. Reducing the number of read ports helps reduce the delay compared to the delay of a single register file.

2.4.4.4 Model Results

Figure 2-27 shows how the model delay results, computed using the constants listed in Appendix B, compare to the Spice results presented earlier. From the graph we can see

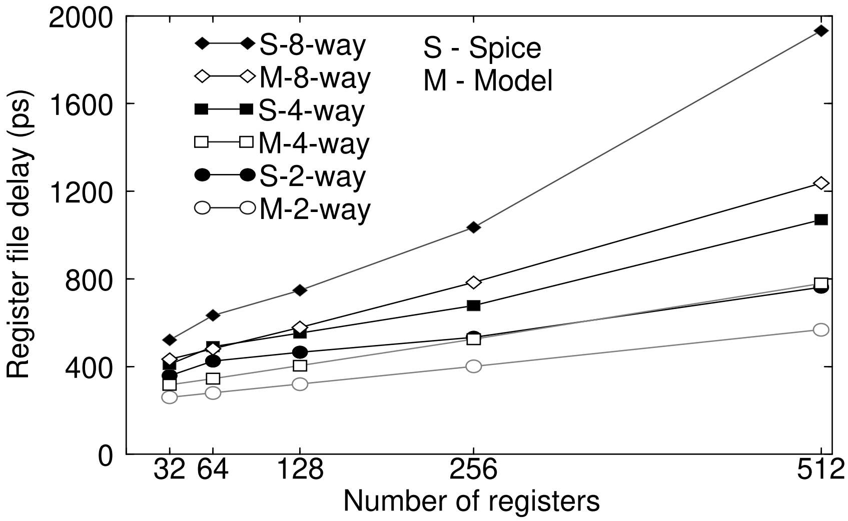


Figure 2-27. Model delay results for register file logic. This graph shows how the model delay results compare to the Spice results for 0.18µm technology.

that the model is successful in tracking the dependence on issue width and register file size.

2.4.5 Data bypass logic

The data bypass logic is responsible for bypassing result values to subsequent instructions that have completed execution but have not yet written their results to the register file. The hardware datapaths and control added for this purpose form the bypass logic. The number of bypasses required is determined by the depth of the pipeline and the issue width of the microarchitecture. As pointed out by Ahuja et al. [ACR95], if IW is the issue width, and if there are S pipestages after the first result-producing stage, then a fully bypassed design would require ($2 \times IW^2 \times S$) bypass paths assuming 2-input functional units. In other words, the number of bypass paths grows quadratically with issue width. The current trend towards deeper pipelines and wider degree of issue only multiplies the number of bypass paths and makes the bypass logic even more critical.

The bypass logic consists of two components: the datapath and the control. The datapath comprises buses, called the result buses, that are used to broadcast bypass values from each source to all possible destinations. The sources of bypass values are the functional units and the cache ports. Buffers are used to drive the bypass values on the result busses. In addition to the result busses, the datapath comprises operand MUXes. Operand MUXes are required to gate in the appropriate results to the operand busses. The fan-in of the operand MUXes is one greater than the number of result busses. The extra input to the MUX is connected to a read port on the register file. This is for cases in which the operand is read from the register file.

The control logic is responsible for controlling the operand MUXes. The control logic compares the tag of the result value to the tag of the source value that is required at each functional unit. If there is a match, the MUX control is set so that the result value is driven on the appropriate operand bus.

The key factor that determines the speed of the bypass logic is the delay of the result wires that are used to transmit bypassed values. The control adds to this delay; however, for our analysis, we will ignore the control because its delay is a small fraction of the total delay. Also, as we move towards smaller feature sizes, wire delays resulting from the result wires will be responsible for a significant fraction of the total delay.

2.4.5.1 Structure

A commonly used structure for the bypass logic is shown in Figure 2-28. The figure shows a bit-slice of the datapath. There are four functional units marked FU0 to FU3. Consider the bit slice of FU0. It gets its two operand bits from the *opd0-l* and *opd0-r* operand wires. The result bit is driven on the *res0* result wire by the result driver. Tristate buffers are used to drive the result bits on the operand wires from the result wires. These buffers implement the MUXes shown in the figure. For example, in order to bypass the result of functional unit FU1 to the left input of functional unit FU0, the tristate driver marked A is switched on. The driver A connects the *res1* wire and *opd0-l* wire. In the case where bypasses are not activated, the operand bits are placed on the operand wires by the

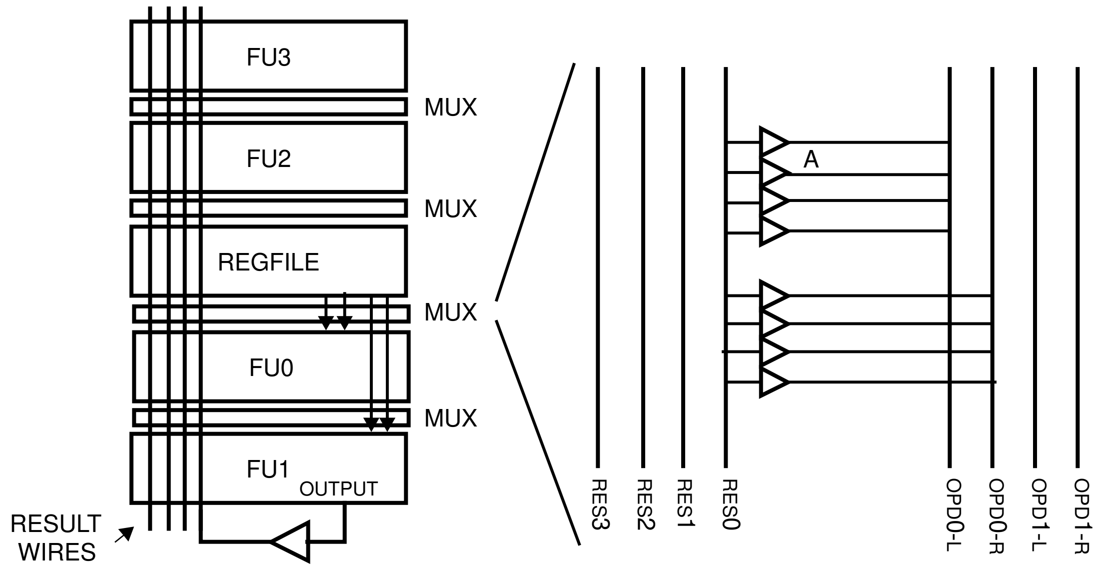


Figure 2-28. Bypass logic.

register file read ports[1]. The result bits are written to the register file in addition to being bypassed.

The delay of the bypass logic is largely determined by the time it takes for the driver at the output of each functional unit to drive the result value on the corresponding result wire. This in turn depends on the length of the result wires. From the figure it can be seen that the length of the result wires is determined by the height of the functional units and the register file. Alternative layouts are possible and are discussed later.

2.4.5.2 Delay Analysis

As discussed before, the delay of the bypass logic can be approximated by the time taken to drive the result bits on the result wires. The equivalent circuit for calculating the delay is shown in Figure 2-29. From the figure it follows,

$$\begin{aligned} T_{delay} &= c_0 \times (R_{driver} \times C_{driver} + (R_{driver} + R_{wire}) \times C_{wire}) \\ &= c_0 \times (R_{driver} \times C_{driver} + (R_{driver} + 0.5 \times R_{metal} \times L) \times C_{metal} \times L) \end{aligned}$$

1. In a reservation-station based microarchitecture the operand bits come from the data field of the reservation station entry.

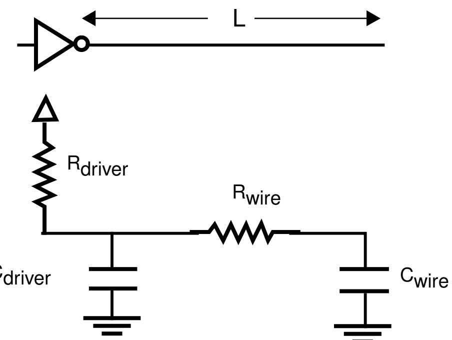


Figure 2-29. Bypass logic equivalent circuit.

where L is the length of the result wires, R_{driver} is the resistance of the pull-up of the driver, and C_{driver} is the diffusion capacitance at the output of the driver. For the layout assumed, the length of the result wires is determined by the height of the functional units and the register file. Each of these terms in turn is a linear function of issue width. Increasing issue width increases the number of functional units and thus lengthens the result wires. Increasing issue width also increases the height of register file because this stretches individual cells in the register file as seen in Section 2.4.4. The result is that the length of the result wires increases linearly with issue width. Rewriting the length of the result wires, L, in terms of issue width, IW, and simplifying we get,

$$T_{delay} = c_0 + c_1 \times IW + c_2 \times IW^2$$

where c_0, c_1, and c_2 are constants. The constants are listed in Table B.14 in Appendix B.

From the above equation we can see that the bypass delay has both a linear component and a quadratic component determined by the issue width. Unlike in the case of other structures, we found that the quadratic component can be significant. Hence, the bypass delay grows quadratically with issue width.

Increasing the depth of the pipeline also increases the delay of the bypass logic as follows. Increasing the depth increases the fan-in of the operand MUXes connected to a given result wire. This in turn increases the amount of capacitance to be charged or dis-

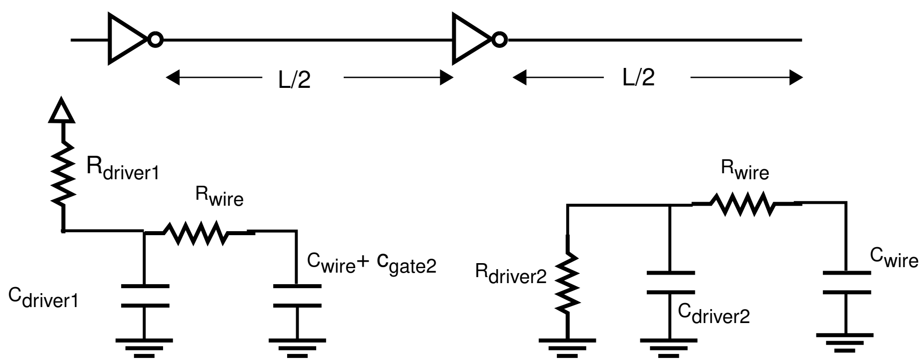


Figure 2-30. Inserting buffers in the result wires.

charged on each result wire because the diffusion capacitance at the output of the operand MUXes adds to the capacitance on the wires. However, this component of the delay is not captured by our simple model. We expect this component of the delay to become relatively less significant as the feature size is reduced.

Buffered result wires

The quadratic component in the delay equation can be reduced in magnitude by inserting buffers in the result wires [WE93]. For example, Figure 2-30 shows the equivalent circuit with a single buffer inserted in each result wire. The resulting delay is given by,

$$\begin{aligned} T_{delay} &= c_0 \times (R_{driver1} \times C_{driver1} + (R_{driver1} + 0.5 \times R_{metal} \times L/2) \times C_{gate2} \\ &+ (R_{driver1} + 0.5 \times R_{metal} \times L/2) \times C_{metal} \times L/2 + R_{driver2} \times C_{driver2} \\ &+ (R_{driver2} + 0.5 \times R_{metal} \times L/2) \times C_{metal} \times L/2) \end{aligned}$$

where $R_{driver1}$ and $C_{driver1}$ are the pull-up resistance and diffusion capacitance of the first driver, $R_{driver2}$ and $C_{driver2}$ are the pull-down resistance and diffusion capacitance of the second driver, and C_{gate2} is the gate capacitance of the second driver. By breaking the result wires in half and inserting a buffer, the magnitude of the quadratic component is reduced in half compared to the unbuffered configuration. However, the total delay will benefit from this reduction only if the delay of the buffer inserted is less than the reduction in the quadratic component. Therefore, this approach of inserting buffers will help reduce delay up to the point where the delay of the inserted buffers equals the delay of each segment of the result wire. Inserting buffers beyond this point will only increase the total delay.

From the delay equation we can see that, even with buffers, the total bypass delay is at least a linear function of issue width. There are two additional factors which augment the criticality of bypass logic. First, bypass logic is in series with the functional units (ALUs) i.e. the sum of ALU delay and bypass delay must be less than the clock period in order to execute dependent instructions in consecutive cycles. As we will see later in Section 2.5, the ability to execute dependent instructions in consecutive cycles is essential for high performance. Second, the result wires are actually longer because in most implementations they extend into the data cache array in order for the cache output to be fanned out to the functional units.

2.4.5.3 Spice Results

We studied the bypass delay for a 2-way, a 4-way and a 8-way machine assuming typical heights for the functional units and the register file. Both buffered and unbuffered result wires were studied. The results are shown in Figure 2-31. There are a number of observations that can be made from the graph. First, the bypass delay increases at least linearly with issue width for both the buffered and unbuffered configurations. For example, assuming unbuffered result wires, the bypass delay increases by factors of 2.4 and 3.0 going from 4-wide to 8-wide issue width for 0.8µm and 0.18µm technology respectively. The increase is higher for 0.18µm technology since the intrinsic wire delay (quadratic) component increases in significance as the feature size is reduced. In fact, for the 0.18µm technology, the intrinsic wire delay is responsible for 68% and 90% of the total delay respectively for the 4-way and the 8-way machine.

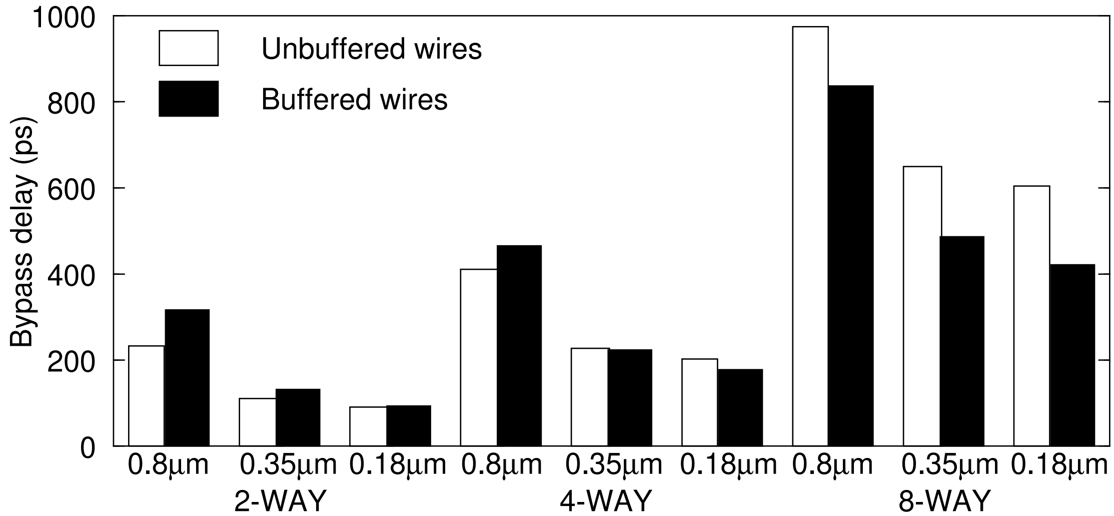


Figure 2-31. Bypass logic delays. For this graph we assume each functional unit has a height of 2500λ, where λ is half the feature size. The length was estimated based on published data [HF88,S+93,I+95]. The height of the register file in each case was computed using the formula, $Height = NPREG \times (cellheight + wordline_{spacing} \times 3 \times IW)$, where *NPREG* is the number of physical registers, *cellheight* is the height of an individual RAM cell excluding the wordline, and $wordline_{spacing}$ is the spacing between wordlines. We use *cellheight* = 24λ, *NPREG* = 48 for 2-way, *NPREG* = 80 for 4-way and *NPREG* = 120 for 8-way, and $wordline_{spacing}$ = 6λ for computing the graph.

Introducing buffers helps mitigate bypass delays for the 8-way machine. For example, now the bypass delay only increases by factors of 1.8 and 2.4 when going from 4-wide to 8-wide issue width for 0.8µm and 0.18µm technology respectively. For the 4-way machine, the reduction is not as significant because the delay of the extra buffer inserted is close to the reduction in the intrinsic delay of the result wire. Another important observation that can be made is that bypass delay does not scale well as the feature size is reduced. For the 8-way machine with buffered result wires, the bypass delay reduces by 42% going from 0.8µm to 0.35µm and by only 13% going from 0.35µm to 0.18µm. This shows that single cycle bypassing between functional units in a wide superscalar machine is going to be increasingly difficult as the feature size is reduced.

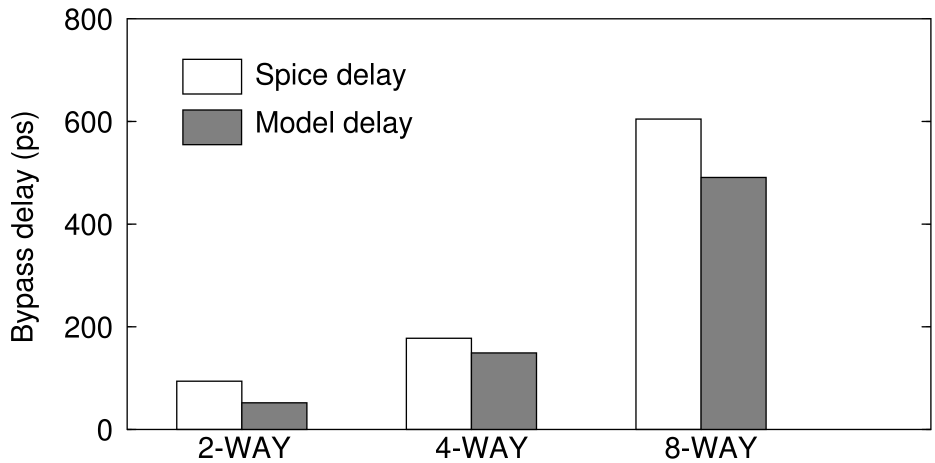


Figure 2-32. Model delay results for bypass logic. This graph shows how the model delay results compare to the Spice results for bypass logic implemented in 0.18μm technology.

2.4.5.4 Model Results

Figure 2-32 shows how the model delay results, computed using the constants listed in Appendix B, compare to the Spice results presented earlier. The model results closely match the Spice results because the bypass delay is dominated by wire delays and our delay models are able to accurately estimate wire delays.

2.4.5.5 Alternative Layouts

The results presented in the previous section assume a particular layout; the functional units are placed on either side of the register file. However, as mentioned before, the length of the result wires is a function of the layout. Hence, microarchitects will have to study alternative layouts in order to reduce bypass delays. Figure 2-33 shows some alternative layouts.

In the alternative shown on the left, all the functional units are placed on one side of the register file. In this case the result wires do not have to extend over the register file. However, the length of the operand wires originating from the register file increases relative to the configuration in Figure 2-28 thus stretching the register file access time. Also, this organization has the disadvantage that the sense amplifiers of the register file cannot be

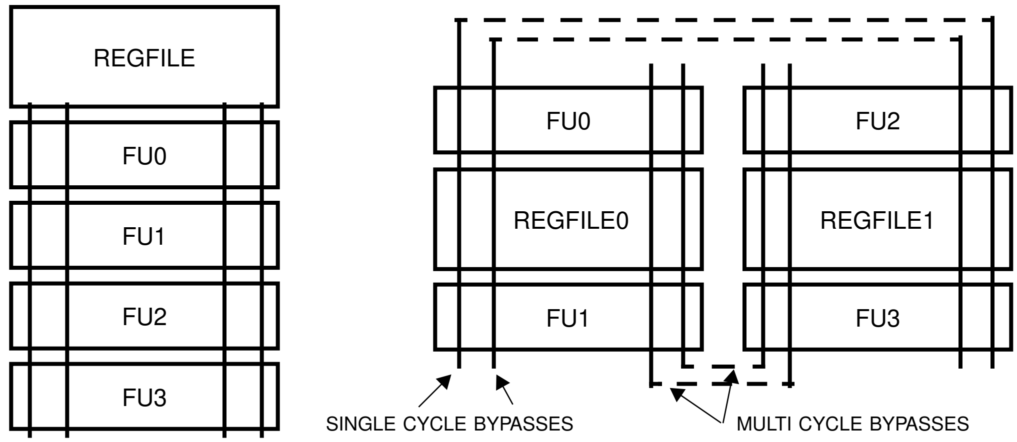


Figure 2-33. Alternative layouts for bypassing.

distributed on both sides. This could stretch the wordlines in the register file and hence, can also increase the register file access time.

In the long term, microarchitects will have to consider clustered organizations like the alternative shown on the right. Each cluster has its own copy of the register file. Bypasses within a cluster complete in a single cycle while inter-cluster bypasses take 2 or more cycles. Such a scheme is implemented in the DEC 21264 [Gwe96a]. The hardware or the compiler or both will have to ensure that inter-cluster bypasses occur infrequently. In addition to mitigating the delay of the bypass logic, this organization also has the advantage of faster register files because there are fewer ports on each register file. Another technique [ACR95] that can be used to improve bypass performance is to use an incomplete bypass network. In an incomplete bypass network only the frequently used bypass paths are provided while interlocks are used in the remaining situations. For an 8-way machine with deep pipelines, this would exclude a large number of bypass paths.

2.5 Pipelining Issues and Overall Delay Results

In the preceding sections, the delay of each of the critical structures was analyzed in detail. However, in addition to the delay, another important consideration is the pipelineability of the structures. Even if the delay of a structure is relatively large it can be eliminated from the critical path defining the clock cycle if it can be pipelined, i.e., its operation is spread over multiple cycles.

However, while deeper pipelining can improve performance by facilitating a faster clock, it can result in a number of side-effects that can degrade performance too. First, the extra stages introduced by deeper pipelining in the front end increase the penalty of mispredicted branches. Also, the penalty of instruction cache misses will increase as a result of extra pipestages that have to be re-filled. At the same time, accurate branch prediction can alleviate these problems to a certain extent. Hence, if the performance improvement achieved as a result of deeper pipelining (faster clock) surpasses the performance degradation caused by the extra stages, then pipelining might be an attractive option. The current trend in the microprocessor industry is towards deeper pipelining. For example, the pipeline in the Intel Pentium Pro [Gwe95b] has as many as 14 pipestages.

The general subject of the effect of pipelining depth on overall performance has been the focus of a number of studies [DF90,JW89,KS86]. We took a different approach in our study. We study the feasibility of pipelining each of the critical structures from the point of view of performance. We identify structures that are amenable to pipelining, i.e., those whose operation can be spread over a small number of pipestages without significantly impacting the IPC factor in the performance equation. Conversely, we identify certain structures that should not be pipelined, especially for programs with limited parallelism, since the pipeline bubbles introduced by pipelining can cause significant degradation in IPCs achieved.

The ability to execute dependent instructions in consecutive cycles is an important requirement for high performance, especially for programs with limited parallelism. The inability to execute dependent instructions back-to-back often introduces pipeline bubbles that can result in significant performance degradation. Experimental results supporting this will be presented later. A simple example will help illustrate this. Consider the time taken to execute a dependent chain of single-cycle instructions of length n. If the ALU operation is segmented into two pipestages, execution of the chain will take $(2 \times n - 1)$ cycles — much more than the n cycles it would take in the non-pipelined case. For this case, even if the clock frequency doubles as a result of pipelining the ALU, overall performance does not improve. In fact, latch overhead could even diminish performance. Even though this example uses an extreme case of zero parallelism, the ability to execute dependent instructions in consecutive cycles is essential, especially for programs with small amounts of parallelism.

Using the ability to execute dependent instructions in consecutive cycles as the metric, we evaluate how pipelining the functions implemented by the individual structures can affect this requirement.

- *Instruction fetching*. Pipelining the instruction fetch logic does not prevent back-to-back execution of dependent instructions. It does, however, increase the penalty of mispredicted branches and instruction cache misses when the pipeline has to be refilled. More accurate branch prediction and an out-of-order back-end help reduce the penalty of mispredicted branches. Our experimental results show that instruction fetching can be pipelined into a few segments at the cost of a small (4% per pipestage) degradation in IPC performance for each extra pipestage introduced. Similar results — 3% per pipestage — have been reported by designers [Hin95] in the industry.
- *Register renaming.* Pipelining register renaming does not prevent back-to-back execution of dependent instructions. Just like instruction fetch, it increases the penalty of mispredicted branches and instruction cache misses and results in similar IPC degradations when pipelined.

  The register rename logic can be pipelined by spreading the dependence checking and the map table access over multiple pipestages. While it is easy to see how dependence checking can be pipelined, it is not so obvious how the map table access can be pipelined. However, there are schemes [Cha91, Now95] for pipelining RAMs that can be applied to map table accesses. In addition, in order to ensure that each rename group sees the map table updates performed by previous rename groups, the updates have to

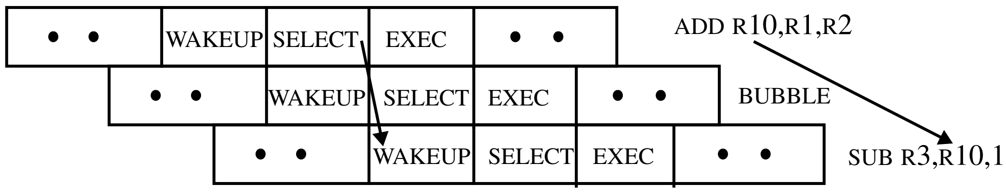


Figure 2-34. Pipelining wakeup and select.

be bypassed around the map table i.e. the updates should be visible before the writes to the table actually complete. Hence, we believe that even though the design will be complicated, register renaming can be pipelined to at least two segments. It must be pointed out that before attempting to pipeline renaming, there are a number of tricks that can be applied to reduce its latency. First, the map table can be duplicated to reduce the number of ports on each copy of the table. Second, because not all instructions have two operands and because it is likely that instructions in a rename group have common operands, the port requirements on the map table can be reduced with little effect on performance.

- *Window logic.* Wakeup and select together have to be accomplished in a single cycle to facilitate back-to-back execution of dependent instructions. If they are spread across multiple pipestages, dependent instructions cannot execute in consecutive cycles as shown in Figure 2-34. The *add* and the *sub* instructions cannot execute back-to-back because the result of the select stage has to feed the wakeup stage. The resulting pipeline bubbles can seriously degrade performance especially in programs with limited parallelism. Hence, wakeup and select together must be accommodated to fit within a cycle.

- *Data bypassing.* Data bypassing is another example of an operation that must be completed in less than a cycle in order to execute dependent instructions in consecutive cycles. The bypass values must be made available to the dependent instruction within a cycle. The delay of the bypass logic is made even more critical by the fact that it is in series with the ALU operation — the sum of the delays of the ALU and the bypass delay should be less than a cycle to facilitate back-to-back execution. As shown earlier, it is going to be increasingly hard to accomplish data bypassing within a single cycle in wide-issue machines.

- *Register file access.* Pipelining the register file does not affect back-to-back execution since the operand values for the consumer instruction are provided by the data bypass logic. Again, like in the case of instruction fetch and register rename, pipelining the register file increases branch mispredict and instruction cache miss penalties. It results in similar IPC degradation as for the case of pipelining front-end stages like instruction fetch and register rename.

  The techniques used to pipeline RAM can be employed to pipeline the register file. Tullsen et al. [T+96] studied the effect of spreading register read over two pipestages. They found that single thread performance degraded by only 2% for their design. Once again, it must be mentioned that instead of pipelining the register file, architects can reduce its latency by duplicating the register file. Each copy of the register file will have half the number of read ports as the original register file. This technique has been used in the DEC 21264 [G+97]. In this case two copies of the integer register file are used.

- *Cache access.* Pipelining cache access can prevent back-to-back execution of dependent instructions. For example, breaking the cache access into two pipeline segments will prevent back-to-back execution of a load instruction and a instruction using the result of the load. In the absence of parallelism, this can severely affect performance. However, cache access is not as critical as window logic or data bypass logic because unlike them, cache access only affects load-use instruction pairs. Pipelining window logic and data bypass logic injects bubbles for all pairs of dependent instructions. While most designs attempt to provide single-cycle cache access, there are designs in which cache access has been pipelined into two stages.

  Caches can be pipelined in a number of ways. One scheme, implemented in the DEC 21064, reads the tags and the data in the first cycle and performs the hit/miss detection operation in the second cycle. A second, more aggressive scheme could pipeline both the tag RAM and the data RAM themselves. A related trade-off is to size the L1 data and instruction caches so that they can be accessed in a single cycle and use a bigger L2 cache to service the L1 misses.

To summarize, the analysis presented above shows that window logic, data bypass logic, and cache access logic implement operations that have to accomplished in a single cycle in order to facilitate execution of dependent instructions in consecutive cycles. Back-to-back execution is a desirable feature from the point of view of performance, especially for codes that have limited parallelism. Because operations that prevent execution of dependent instructions in consecutive cycles will not be pipelined for performance reasons, we believe that the latency of these operations will ultimately limit the degree of pipelining. Consequently, the delays of these operations are crucial and will determine the complexity of a microarchitecture.

The qualitative analysis presented above is not new. Similar issues and trade-offs have been discussed in the context of deep pipelining [KS86] and superpipelining [JW89]. The trade-offs are analyzed here in the context of out-of-order microarchitectures. The move towards wide-issue superscalar machines and the technology trend of wire delays dominating total delays increases the importance of these trade-offs and hence, architects need to reevaluate these trade-offs. There are a few caveats to the analysis. The arguments presented are tightly hinged on the assumption that there is limited instruction-level parallelism in programs. At least theoretically, in the hypothetical situation of very high-levels of parallelism, pipelining any of the structures will not significantly impact performance. Also, pipelining cannot be used as a panacea for reducing complexity. Pipelining, especially deep pipelining, has its own set of drawbacks. Clock skew and latch overhead can combine to limit the decrease in clock period obtained by further pipelining. Deep pipelining also requires sophisticated circuit design.

To quantify the effect of pipelining the above operations on the effectiveness of a microarchitecture, we studied the performance effect of varying the number of pipeline

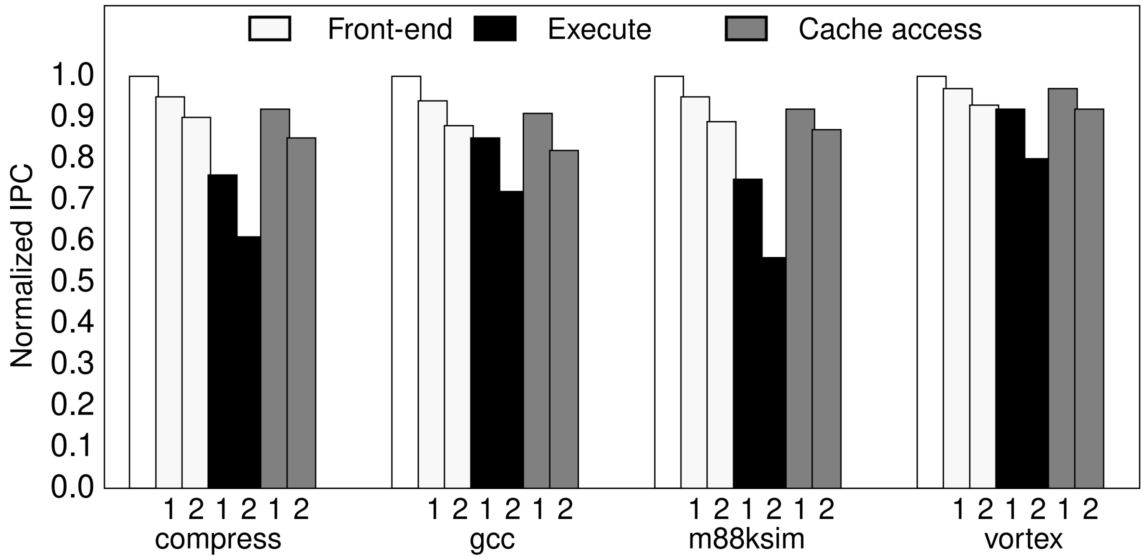


Figure 2-35. Effect of pipelining on IPC. This figure shows the effect of pipelining on the performance of a 8-way out-of-order microarchitecture. Each pair of bars shows the effect of introducing one (1) and two (2) extra pipestages in that particular section. The leftmost bar for each benchmark shows the base performance. The simulated processor has a 64-entry window, a 120-entry register file, and a gshare branch predictor with 20 bits of global history.

stages. A baseline out-of-order microarchitecture of the kind shown in Figure 2-1 on page 15 is assumed. The pipeline was divided into three sections: front-end, execute, and cache access. The front-end section includes instruction fetch, register rename, and register file access operations. The execute section includes window wakeup, window selection, and data bypass operations. The cache access section consists of only the cache access operation. The pipeline was partitioned in this fashion because the operations in a given section are identical with respect to pipelining i.e. spreading register rename over two stages and spreading register file access over two stages have the same effect on performance. We then studied the effect of introducing extra stages in each section. The results are shown in Figure 2-35.

A number of observations can be made from the graph. First, pipelining the operations in the front-end does not degrade effectiveness significantly. However, pipelining the operations in the execute section can result in serious performance degradation and hence, should be avoided. For example, dividing execute into two pipestages can degrade the per-

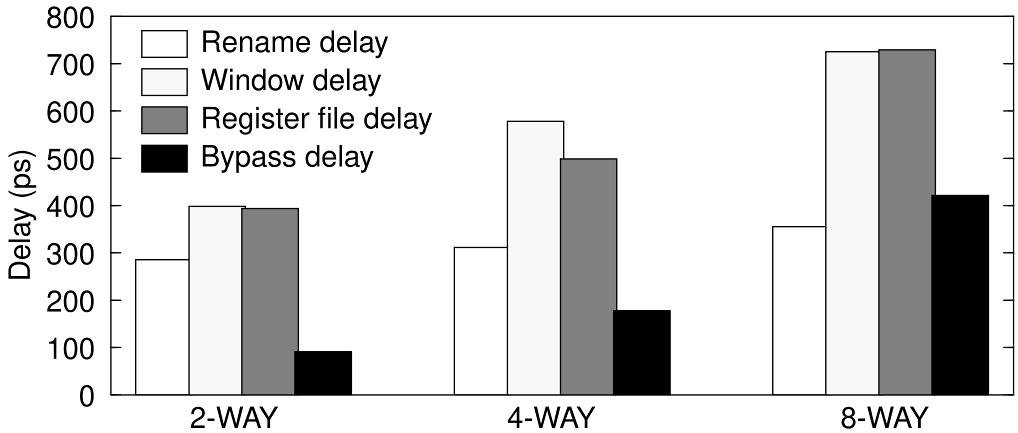


Figure 2-36. Overall delay results. This figure shows overall delay results for a 2-way, a 4-way, and a 8-way machine in 0.18µm technology. The 2-way machine has a 48-entry register file and a 16-entry window. The 4-way machine has a 80-entry register file and a 32-entry window. The 8-way machine has a 120-entry register file and a 64-entry window.

formance by as much as 24% in the case of *compress*. Pipelining cache access can degrade performance by as much as 8% per additional pipestage[1]. Hence, it is important to keep the cache access latency low (less than 3 cycles) for good performance. In summary, the graph shows that while register file access and register renaming can be pipelined without taking a significant hit in performance, performing window and data bypass operations in a single cycle is crucial for high performance.

Overall delay results

The overall delay results for a 2-way, a 4-way, and a 8-way microarchitecture in 0.18µm technology are shown in Figure 2-36. The corresponding results for 0.8µm and 0.35µm technologies are shown in Appendix A. The graph shows that the delay of window logic, register file logic, and data bypass logic increases significantly with issue width. The data bypass logic shows the largest increase, increasing by factors of 1.95 and 2.37 going from 2-way to 4-way and from 4-way to 8-way respectively. Even though the delay of the bypass logic is smaller than that of the window logic and the register file logic, the fact that the bypass logic is in series with the functional units makes its performance critical.

1. Wilson and Olukotun [WO95] report similar numbers.

Another observation is that when the issue width is increased from 4 to 8, the register file delay degrades more than the window delay. This is explained by the particular configurations assumed for the graph. The size of the register file increases from 80 registers to 120 registers whereas the window size only increases from 32 entries to 64 entries. Also, the delay of the window selection logic is the same for both the configurations because selection logic increases logarithmically (base 4) with window size. However, as shown earlier, the register file logic is not as critical as the window logic because it can be pipelined with a small reduction in IPC. Hence, window logic and bypass logic are the most crucial structures among the list of structures studied here.

2.6 Related Work

The access time of caches and register files have been studied in the past. Wada et al. [WRP92] quantify the access time of a cache as a function of various cache parameters like cache size, associativity, and line size. Wilton and Jouppi [WJ94] further refined Wada's model. The methodology used for this work is similar to the one used by Wilton and Jouppi [WJ94]. Farkas et al. [FJC96] modified the cache model to study, using Spice simulations, how the access time of a register file varies with the size of the register file and the number of ports. In this chapter we develop analytical equations in addition to presenting Spice simulation results for register files. Specific implementations of register files are described in published literature [AMG+95,Jol91, S+91].

The subject of quantifying the complexity of issue logic in superscalar microarchitectures has received some attention, mostly qualitative, in the past. Horowitz et al. [HPS92] argue that increasing complexity, both due to worsening wire delays and growing interconnection cost, will ultimately limit the performance advantage of wide-issue, dynamically scheduled, superscalar microarchitectures. They measure complexity of a specific operation in terms of the number of gates, or in some cases the die area, required for implementing the operation. For example, they show that the complexity of operand bypassing grows as $O(IW^2)$ where IW is the issue width.

Johnson [Joh91] gives a largely qualitative description of the complexity of a central window. He points out that the critical path for the window logic comprises two operations: an instruction being made ready by a result that will be forwarded and the arbitration for a functional unit by that instruction, i.e. a wakeup followed by a select in our terminology. Based on some assumptions, he estimates that the critical path can be implemented using 16 logic stages. He does not consider wire delays in his analysis.

Chamdani [Cha95] measures the complexity of superscalar microarchitectures in terms of hardware costs. He presents a theoretical cost analysis in terms of the costs of a 1-bit comparator, 1-bit register storage, 1-bit global wire, and other unit parameters. The cost analysis is used to compare various superscalar designs.

There are a number of studies that discuss how interconnect delays can become a significant limiter in future technologies. Bohr claims [Boh95] that as clock frequencies approach 1 GHz and interconnect pitches shrink below 0.5μm, interconnect delay will become a dominant portion of clock cycle time. Even though increasing metal aspect ratio helps improve RC delay, he shows that maximum benefits are achieved once aspect ratios reach close to 2. Furthermore, the study also shows that using more interconnect layers is not a feasible solution since the practical limits for the number of layers will be reached in just a few technology generations. Wilhelm [Wil95] presents a lucid explanation, starting from basic principles, of the poor scaling of wire delays in future. He concludes that the impending wire delay problem will force architects to consider designs that avoid global signalling. Matzke [Mat97] introduces the notion of *signal drive region* and *clock locality matrix* to show how multiple clock cycles will be required to propagate signals across a die in future. He also concludes that only microarchitectures with good locality and corresponding floor planning will survive.

2.7 Chapter Summary

This chapter analyzed the delay of critical structures in a baseline superscalar microarchitecture. The structures studied are critical in the sense that their delay is a function of issue width, issue window size, wire delays and hence, it is likely that the delay of these structures will determine the cycle time in future designs in advanced technologies. Simple analytical models that expressed the delay of each of the structures in terms of microarchitectural parameters like issue width and instruction window size were developed. In addition, we studied how the delays scale as feature sizes shrink and wire delays become more prominent.

The overall results show that the logic associated with managing the issue window of a superscalar processor is likely to become the most critical structure as we move towards wider issue, larger windows, and advanced technologies in which wire delays dominate. One of the functions implemented by the window logic is the broadcast of result tags to all the waiting instructions in the window. The delay of this operation is determined by the delay of wires that span the issue window. We found that the delay of this operation increases at least linearly with window size and issue width. Hence, this operation does not scale well. Furthermore, in order to be able to execute dependent instructions in consecutive cycles, the delay of the window logic should fit within a cycle.

In addition to the window logic, a second structure that needs careful consideration especially in future technologies is the data bypass logic. The length of the result wires used to broadcast bypass values increases linearly with issue width and hence, the delay of the data bypass logic increases at least linearly with issue width. Again, in order to be able to execute dependent instructions in consecutive cycles, the sum of the delay of a functional unit and the data bypass logic should be less than a cycle.

In summary, straightforward scaling of current microarchitectures will not be sufficient because the resulting wire delays could significantly impact cycle time thus reducing the overall performance improvement. As wire delays increasingly dominate total delay, architects have to design more compact microarchitectures that avoid global signalling.

Chapter 3

Dependence-based Superscalar Microarchitectures

The analysis presented in Chapter 2 shows that issue window logic is one of the primary contributors of complexity in a conventional microarchitecture. The delay of the window logic increases at least linearly with both issue width and window size. Furthermore, the wakeup and select operations implemented by the window need to be accomplished in a single cycle for high performance. In addition to window logic, another structure whose delay scales poorly with issue width, especially in future technologies, is the data bypass logic. The length of the result wires used to broadcast bypass values increases linearly with the number of functional units and hence, the delay of data bypass logic grows at least linearly with issue width. This chapter proposes and evaluates dependence-based superscalar microarchitectures that address the complexity of the window logic and the data bypass logic. The proposed microarchitectures are designed to extract similar levels of parallelism as conventional microarchitectures while enabling a faster clock.

Dependence-based microarchitectures use two main techniques to achieve the dual goals of wide-issue and fast clock. *Partitioning* is used to enable a fast clock. The machine is partitioned into multiple clusters each of which contains a part of the instruction window and the execution resources of the whole processor. This enables high-speed clocking of the clusters since the narrow issue width and the small instruction window of each cluster keeps the critical delays small. The second technique is applied to sustain a high IPC for the whole machine. This involves *intelligent steering* of instructions to the multiple clusters so that the full width of the machine is utilized while minimizing the performance degradation due to slow inter-cluster communication. Dependences between instructions, discovered at run-time, are used as input to perform the steering. Hence, the name *dependence-based* superscalar microarchitectures. It must be pointed out that the two techniques must be used in conjunction since both a fast clock and a high IPC are necessary for high performance.

The rest of this chapter is organized as follows. The next section discusses the concept behind the dependence-based superscalar microarchitectures. Section 3.2 presents and analyzes in detail a specific instance of the dependence-based superscalar microarchitectures called the *fifo-based microarchitecture*. Section 3.3 discusses other interesting members of the family of dependence-based microarchitectures. Experimental evaluation results are presented in Section 3.4. Other related microarchitectures are discussed in Section 3.5, and finally, the chapter is summarized in Section 3.6.

3.1 Concept

The organization of a generic dependence-based superscalar microarchitecture is illustrated in Figure 3-1. The issue and execution resources of the machine are partitioned into multiple clusters. Renamed instructions are steered to one of the clusters. Steering issues are discussed later. Each cluster contains a slice of the instruction window and the functional units of the whole machine. A copy of the register file is provided in each cluster. The multiple copies of the register file are kept identical by broadcasting register writes. Local bypasses within a cluster (shown using thick lines) are responsible for bypassing values produced in a given cluster to the inputs of the functional units in the same cluster. By keeping the issue width of the clusters small, local bypassing is accomplished in a sin-

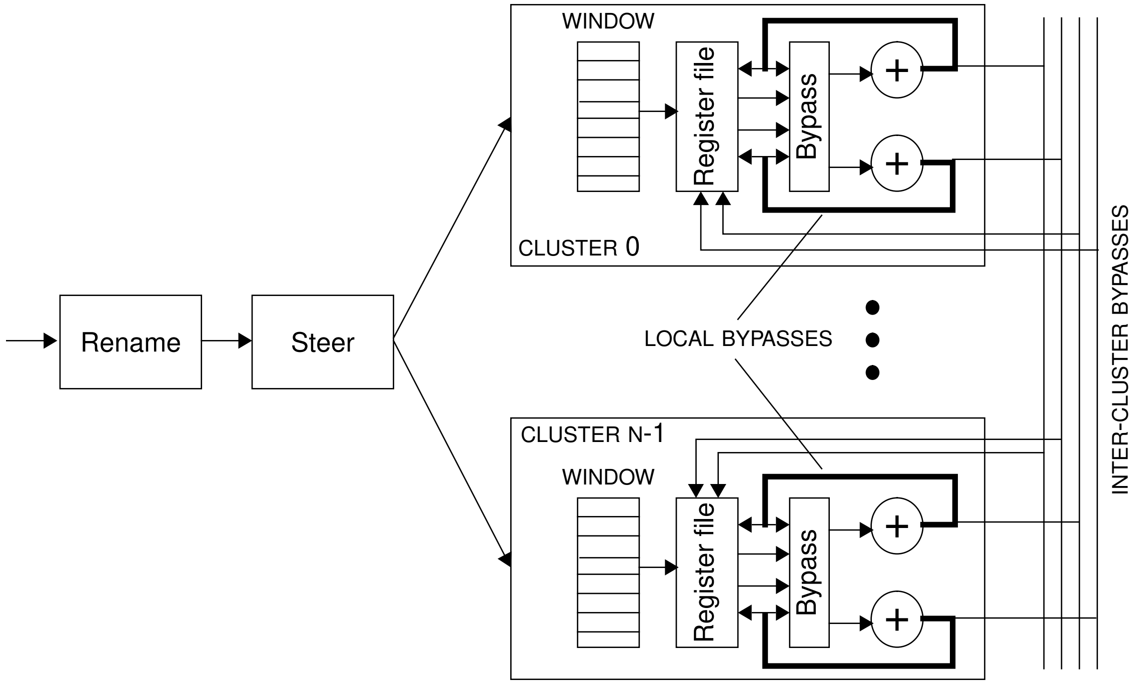


Figure 3-1. Dependence-based superscalar microarchitecture.

gle cycle. Inter-cluster bypasses are responsible for bypassing values between functional units residing in different clusters. Because inter-cluster bypasses require long wires, it is likely that these bypasses will be relatively slower and take two or more cycles in future technologies. The inter-cluster bypass wires are also used to keep the multiple copies of the register file coherent. Hence, the multiple copies are identical except for the one or more cycles difference in propagating results from one cluster to the rest of the clusters.

The proposed microarchitecture has a number of advantages over the conventional microarchitecture with respect to complexity. Since each cluster implements a narrow execution core with a small window, both the window logic and data bypass logic delays in each cluster can be kept small. As a result, the proposed microarchitecture can support a faster clock than a wide conventional microarchitecture with a large issue window. Also, by using multiple copies of the register file, the dependence-based microarchitecture reduces the number of ports on the register file and makes the access time of the file faster, relative to that of a centralized file.

The front-end of the dependence-based superscalar microarchitecture is identical to that of the conventional microarchitecture except for the addition of steering logic. The steering logic is responsible for steering instructions to individual clusters based on dependences extracted at run-time. The goal of the steering logic is to make use of the full width of the machine while minimizing the use of slow inter-cluster communication. Even though the figure shows the steering logic to be in series with the rename logic, simple versions of the steering logic can be implemented to operate in parallel with the rename logic, thus eliminating the need for an extra pipestage. Section 3.3.3 discusses the trade-offs involved in more detail.

Since the proposed microarchitecture uses the same front-end as a conventional microarchitecture, it does not reduce the complexity of instruction fetch and renaming. Extra pipestages, at the expense of a reduction in IPC as shown in Section 2.5 in Chapter 2, is one way to reduce the complexity of the front-end.

Performance factors

The overall performance of a dependence-based microarchitecture is highly dependent on the amount of ILP that can be extracted relative to the conventional microarchitecture. If the microarchitecture can sustain comparable IPCs, then its clock speed advantage will result in higher overall performance. The primary factors that determine the IPCs achieved by the proposed microarchitecture are:

- *Load balancing*. It is important that instructions are spread out to use as many clusters as the amount of program parallelism allows. Otherwise, the program will not be able to take advantage of the full-width of the machine. For example, if we have a 8-way dependence-based superscalar microarchitecture organized as 4 clusters each being 2-wide, and if all instructions are steered to a single cluster, the machine will be effectively reduced to a 2-wide machine.
- *Inter-cluster bypass frequency.* Since inter-cluster communication is slow, excessively using the inter-cluster bypass paths can easily stretch the critical path of the program,

resulting in poor performance. Hence, it is essential that the steering logic minimize the frequency of inter-cluster bypasses exercised. It must be pointed out that inter-cluster bypass frequency must be judged along with load balancing. For example, it is possible to completely eliminate inter-cluster communication by steering all instructions to a single cluster. However, performance can be significantly degraded because of the reduced effective width of the machine. Hence, the challenge is to be able to balance the load across multiple clusters while minimizing the frequency of inter-cluster bypasses.

- *Steering logic complexity.* Complex steering logic will require multiple pipestages that can result in IPC degradation due to increase in penalties associated with branch mispredicts and instruction-cache misses. This can reduce the benefit of achieving good load balance and minimizing inter-cluster bypass frequency. Hence, the steering logic must be kept simple.

The results presented in the rest of the chapter will show that it is possible to achieve good steering with simple steering heuristics.

3.2 Dependence-based Microarchitectures : An Example

This section describes a particular dependence-based microarchitecture called the *fifo-based* microarchitecture. The idea behind the fifo-based microarchitecture is to exploit the natural dependences among instructions. A key point is that dependent instructions cannot execute in parallel. In a single-cluster version of the proposed microarchitecture, shown in Figure 3-2, the issue window is replaced by a small number of fifo buffers. The fifo buffers are constrained to issue in-order, and dependent instructions are steered to the same fifo. This ensures that instructions in a particular fifo buffer can only execute sequentially. Hence, unlike the typical issue window where result tags have to be broadcast to all the entries, the register availability only needs to be fanned out to the heads of the fifo buffers. The instructions at the fifo heads monitor reservation bits (one per physical register) to

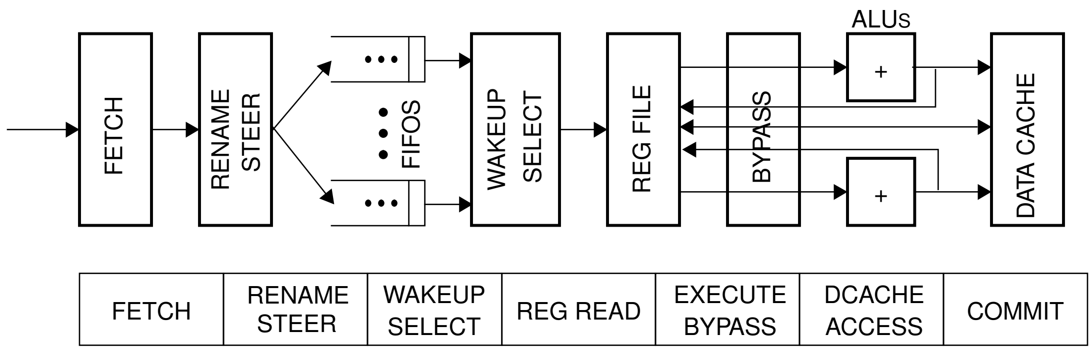


Figure 3-2. Fifo-based microarchitecture.

check for operand availability. This is discussed in detail later. Furthermore, the selection logic only has to monitor instructions at the heads of the fifo buffers.

The steering of dependent instructions to the fifo buffers is performed at run-time during the rename stage. Dependence information between instructions is maintained in a table called the SRC_FIFO table. This table is indexed using logical register designators. For example, SRC_FIFO[Ra], the entry for logical register Ra, stores the identity of the fifo buffer containing the instruction that will write register Ra. If that instruction has already completed i.e. register Ra contains its computed value, then SRC_FIFO[Ra] is invalid. This table can be accessed in parallel with the rename table. In order to steer an instruction to a particular fifo, the SRC_FIFO table is accessed with the register identifiers of the source operands of an instruction. For example, to steer the instruction `add r10,r5,1` where `r10` is the destination register, the SRC_FIFO table is indexed with 5. The entry is then used to steer the instruction to the appropriate fifo.

A number of heuristics are possible for steering instructions to the fifos. A simple heuristic that we found to work well for our benchmark programs is described next. Let *I* be the instruction under consideration. Depending upon the availability of *I*'s operands, the following cases are possible:

1. *All operands available*. All the operands of *I* have already been computed and are residing in the register file. In this case, *I* is steered to a new (empty) fifo acquired from

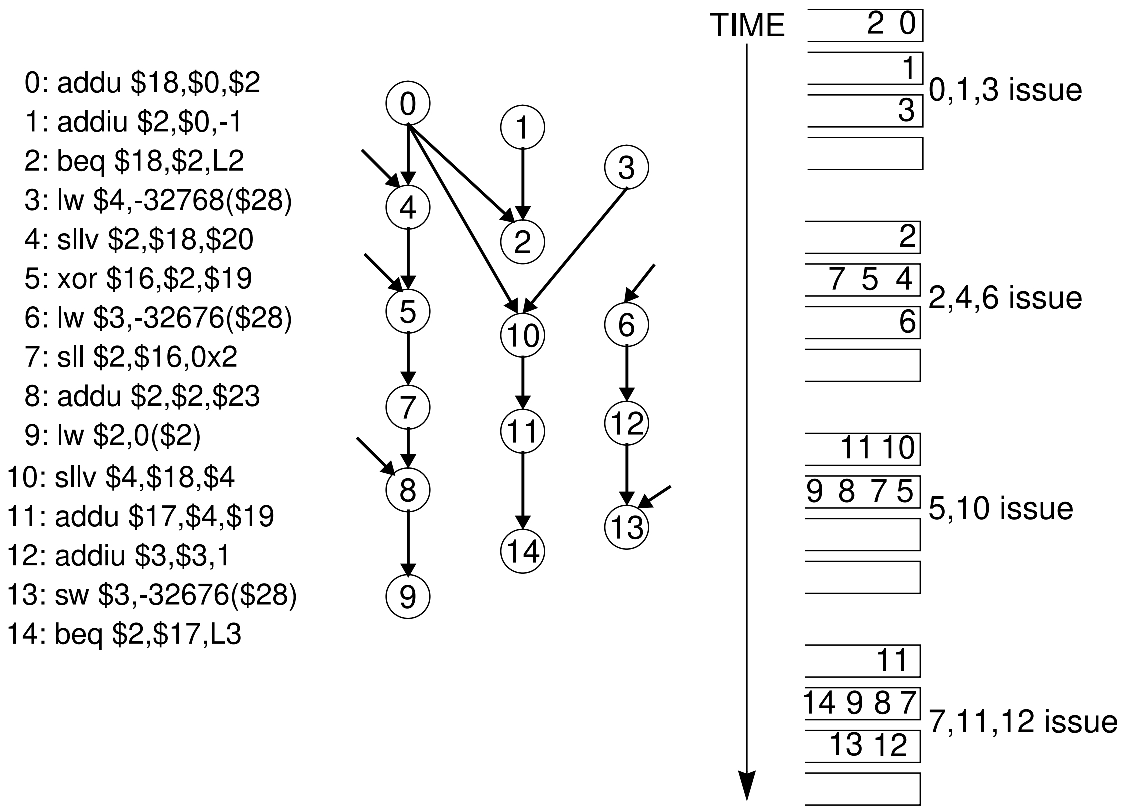


Figure 3-3. Instruction steering example.

a pool of free fifos.

2. *One outstanding operand. I* requires a single outstanding operand to be produced by instruction I_{source} residing in fifo F_a. In this case, if there is no instruction behind I_{source} in F_a, then *I* is steered to F_a, else *I* is steered to a new fifo.
3. *Two outstanding operands. I* requires two outstanding operands to be produced by instructions I_{left} and I_{right} residing in fifos F_a and F_b respectively. In this case, apply the heuristic in the previous bullet to the left operand. If the resulting fifo is not suitable (it is either full or there is an instruction behind the source instruction), then apply the same heuristic to the right operand.

If all the fifos are full or if no empty fifo is available then the steering logic stalls. A fifo is returned to the free pool when the last instruction in the fifo is issued. Initially, all the fifos are in the free pool. Figure 3-3 illustrates the heuristic on a code segment from the

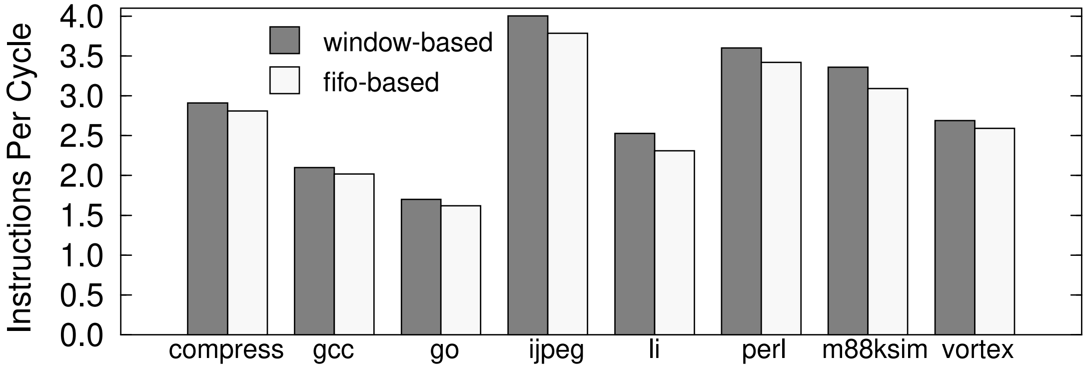


Figure 3-4. Performance of single-cluster fifo-based microarchitecture.

SPEC benchmark *compress* for a 4-wide machine. The listing on the left shows the dynamic stream of instructions. The directed graph in the middle shows the register dependences between those instructions. On the right side of the figure are the contents of the fifos in each cycle. Instructions can issue only from the heads of the four fifos. The steering logic steers four instructions every cycle and a maximum of four instructions can issue every cycle. Consider the steering performed in cycle 1. Instructions 4, 5, 6, and 7 are steered to the appropriate fifos. Since instructions 4, 5, and 7 form a dependence chain, they are steered to the same fifo. Because instruction 6 is a ready instruction (which happens to start a dependence chain) it is steered to a new fifo. In the next cycle, instructions 8, 9, 10, and 11 are steered. Since instructions 8 and 9 form a chain that depends on instruction 7, they are steered to the fifo containing instruction 7. Similarly, instructions 10 and 11 form a chain and are steered to a new fifo.

3.2.1 Performance of the Fifo-based Microarchitecture

Comparison with window-based superscalar

We compare the performance of the fifo-based microarchitecture against that of a typical microarchitecture with a single, large issue window. The proposed microarchitecture has 8 fifos, with each fifo having 8 entries. The issue window of the conventional processor has 64 entries. Both microarchitectures can decode, rename, and execute a maximum of 8 instructions every cycle. The simulation model assumed is detailed in Table 3.2 on page 105.

The performance results in terms of instructions committed per cycle are shown in Figure 3-4. The fifo-based microarchitecture extracts similar parallelism as the typical window-based microarchitecture. The cycle count numbers are within 5% for five of the seven benchmarks and the maximum performance degradation is 8.7% in the case of *perl*.

Fifo utilization

The graph on the left in Figure 3-5 shows the time distribution of the number of active fifos during the execution of *m88ksim*. A fifo is active if it contains at least one instruction. While the graph shows that for a majority of the time all the fifos are utilized, there are periods during which fewer fifos are active. This shows that the distribution of parallelism in the program is uneven — there are phases in which the average number of parallel chains is small. Other benchmarks show similar results.

The graph on the right in Figure 3-5 shows the time distribution of the depth of a particular fifo during the execution of *m88ksim*. The graph shows that on average the number of instructions in a fifo is small. This is for two reasons. First, the steering heuristic stalls whenever a suitable fifo is not found. We found that placing the stalled instruction in a random fifo could degrade performance for certain programs. Second, and more importantly, frequent branch mispredicts cause breaks in the instruction stream presented to the steering logic, resulting in shallow fifos on the average. We found similar distributions for the other benchmarks.

Effect of increasing number of fifos

Increasing the number of fifos increased the performance for all the benchmarks. However, the improvements were in the 2%-3% range for as many as 12 fifos. Eight fifos are able to support most of the parallel chains found at any instance during the execution of the programs.

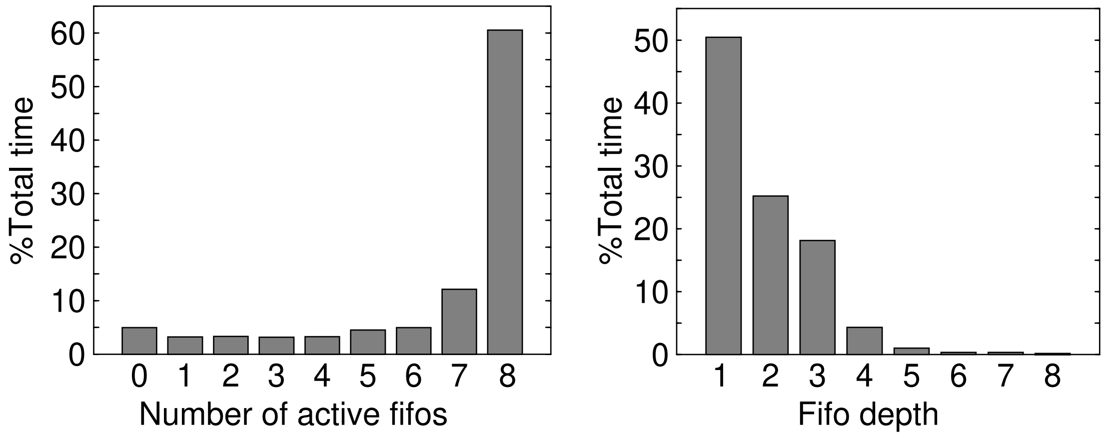


Figure 3-5. Fifo utilization. The graph on the left shows the number of active fifos during the execution of *m88ksim*. The graph on the right shows the depth of a particular fifo during the execution of the program.

3.2.2 Complexity Analysis of the Fifo-based Microarchitecture

First, consider the delay of the wakeup and selection logic. Wakeup logic is required to detect cross-fifo dependences. For example, if the instruction I_a at the head of fifo F_a is dependent on an instruction I_b waiting in fifo F_b, then I_a cannot issue until I_b completes. However, the wakeup logic does not involve broadcasting result tags to all the waiting instructions. Instead, only the instructions at the fifo heads have to determine when all their operands are available. This is accomplished by interrogating a table called the reservation table. The reservation table contains a single bit per physical register that indicates whether the register is waiting for its data. When an instruction is dispatched, the reservation bit corresponding to the physical register is set. The bit is cleared when the instruction executes and the result value is produced. An instruction at the fifo head waits until the reservation bits corresponding to its operands are cleared. Hence, the delay of the wakeup logic is determined by the delay of accessing the reservation table. The reservation table is relatively small in size compared to the rename table and register file. For example, for a 4-way machine with 80 physical registers, the reservation table can be laid out as a 10-entry table with each entry storing 8 bits. A column MUX is used to select the appropriate bit from each entry. Table 3.1 shows the delay of the reservation table for 4-way and 8-

Issue width	# physical regs	# table entries	Bits/entry	Delay(ps)
4	80	10	8	192.1
8	128	16	8	251.7

Table 3.1: Delay of reservation table in 0.18µm technology.

way machines. For both cases, the wakeup delay is much smaller than the wakeup delay for a 4-way, 32-entry issue window-based microarchitecture. Also, this delay is smaller than the corresponding register renaming delay. The selection logic in the fifos dependence-based microarchitecture is simple because only the instructions at the fifo heads need to be considered for selection.

Instruction steering is done in parallel with register renaming. Because the SRC_FIFO table is smaller than the rename table we expect the delay of steering to be less than the rename delay. In case a more complex steering heuristic is used, the extra delay can easily be moved into the wakeup/select stage or a new pipestage can be introduced — at the cost of an increase in the branch mispredict and instruction-cache miss penalties.

In summary, the complexity analysis presented above shows that by reducing the delay of the window logic significantly, it is likely that the fifo-based microarchitecture can be clocked faster than the typical microarchitecture. Combining the potential for a much faster clock with the results indicate the dependence-based microarchitecture is capable of superior performance relative to a conventional microarchitecture.

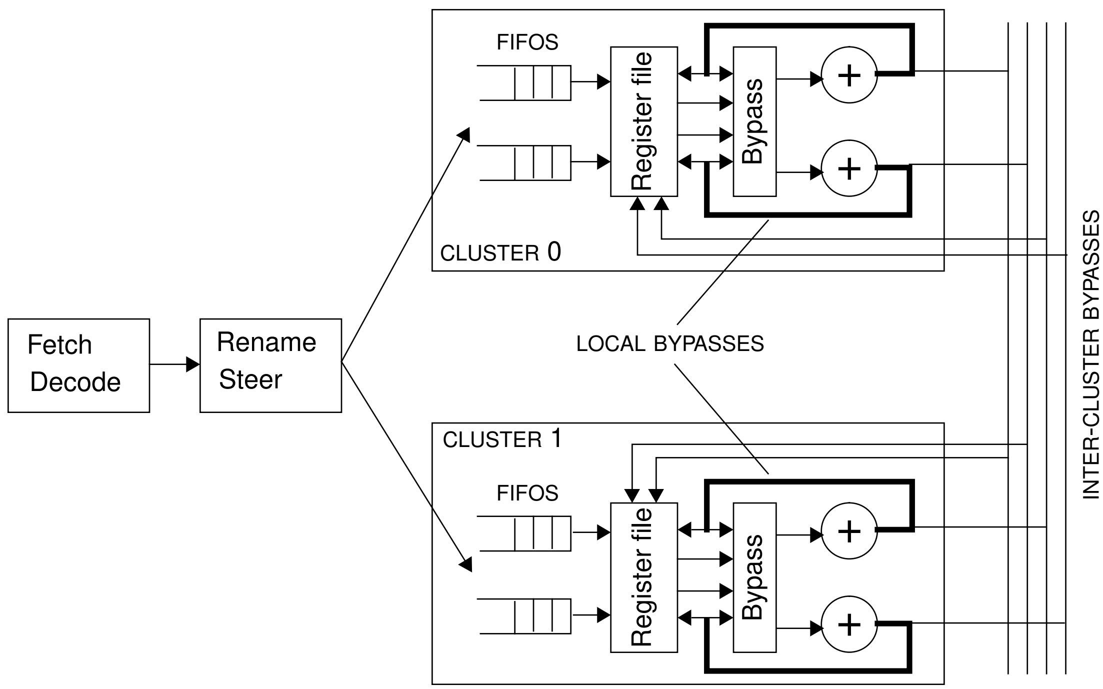


Figure 3-6. Fifo-based microarchitecture with two clusters.

3.2.3 Clustering the Fifo-based Microarchitecture

The real advantage of the fifo-based microarchitecture is for building machines with issue widths greater than four where, as shown in the previous chapter, the delay of both the large window and the long bypass busses can be significant and can considerably slow the clock. Dependence-based microarchitectures based on fifos are ideally suited for such situations because they simplify both the window logic and the bypass logic as well as naturally facilitate efficient steering. Such a microarchitecture for building an 8-way machine is described next.

Consider the 2X4-way clustered system shown in Figure 3-6. Two clusters are used, each of which contains four fifos, one copy of the register file, and four functional units. Renamed instructions are steered to a fifo in one of the two clusters. Local bypasses (shown using thick lines) permit same-cycle bypassing inside each cluster. Local bypassing can be accomplished within a cycle. Inter-cluster bypasses, responsible for bypassing

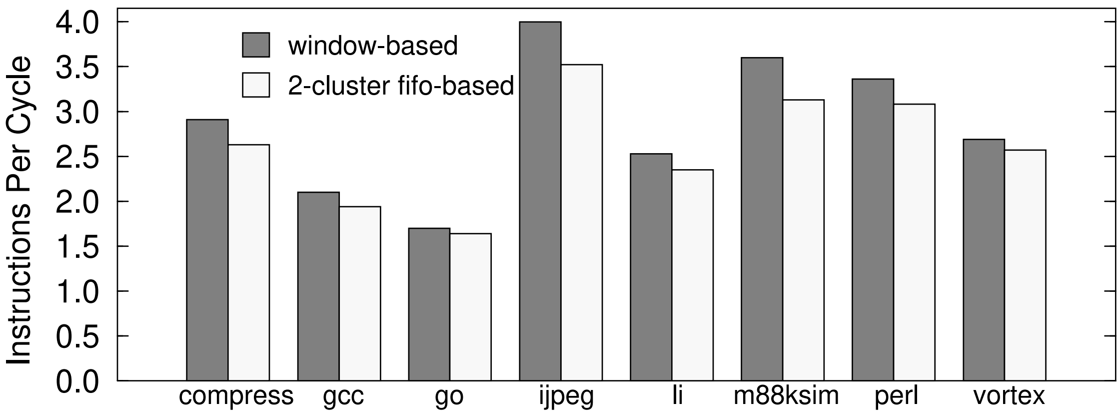


Figure 3-7. Performance of the clustered fifo-based microarchitecture.

values between functional units residing in different clusters, take one or more additional cycles.

This dependence-based microarchitecture using fifos has a number of advantages. First, wakeup and selection logic are simplified as noted previously. Second, because of the heuristic for assigning dependent instructions to fifos, and, indirectly, to clusters, local bypasses are used much more frequently than inter-cluster bypasses, reducing overall bypass delays.

3.2.4 Overall Performance of the Clustered Fifo-based Microarchitecture

The graph on the left in Figure 3-7 compares performance, in terms of instructions committed per cycle (IPC), for the 2X4-way dependence-based microarchitecture against that of a conventional 8-way microarchitecture with a single 64-entry issue window. For the dependence-based microarchitecture, instructions are steered using the heuristic described in Section 3.2. Local bypasses complete within a cycle while inter-cluster bypasses take 2 cycles. Also, in the conventional 8-way system all bypasses are assumed to complete in a single cycle. From the graph we can see that for most of the benchmarks, the dependence-based microarchitecture is nearly as effective as the window-based microarchitecture even though the dependence-based microarchitecture is handicapped by slow inter-cluster bypasses that take 2 cycles. However, for two of the benchmarks, *m88ksim* and *ijpeg*, the

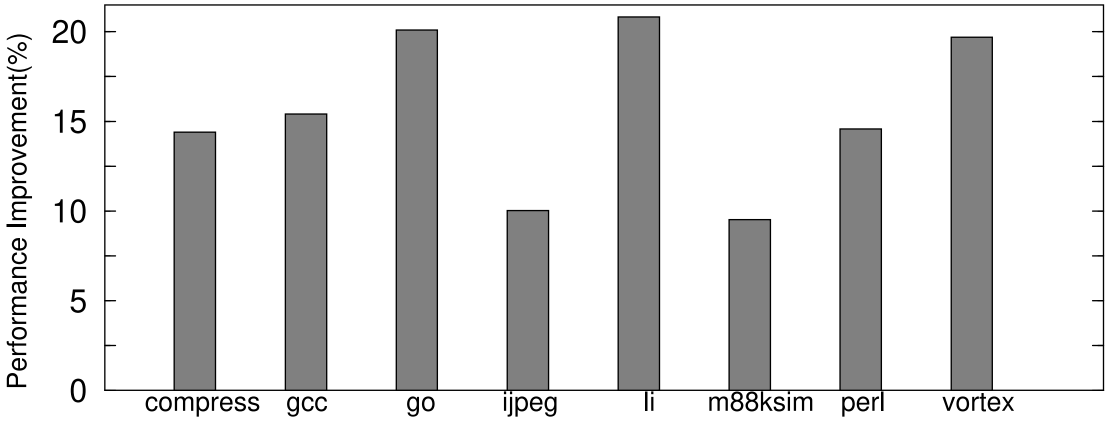


Figure 3-8. Potential improvements with the fifo-based microarchitecture.

performance degradation is close to 13%. We found that this degradation is mainly due to extra latency introduced by the slow inter-cluster bypasses.

Because the dependence-based microarchitecture will facilitate a faster clock, a fair performance comparison must take clock speed into account. The local bypass structure within a cluster is equivalent to a conventional 4-way superscalar machine, and inter-cluster bypasses are removed from the critical path by taking an extra clock cycle. Consequently, the clock speed of the dependence-based microarchitecture is at least as fast as the clock speed of a 4-way, 32 entry window-based microarchitecture, and is likely to be significantly faster because of the smaller (wakeup + selection) delay compared to a conventional issue window as discussed in Section 3.2.2. Hence, if C_{dep} is the clock speed of the dependence-based microarchitecture and C_{win} is the clock speed of the window-based microarchitecture then from Table A.10 in Appendix A for 0.18µm technology,

$$\frac{C_{dep}}{C_{win}} \geq \frac{\textit{delay of 8-way 64-entry window}}{\textit{delay of 4-way 32-entry window}} = 1.25$$

In other words, the dependence-based microarchitecture is capable of supporting a clock that is 25% faster than the clock of the window-based microarchitecture. Taking this factor into account (and ignoring other pipestages that may have to be more deeply pipelined), we can estimate the potential speedup with a dependence-based microarchitecture. The speedups for the benchmarks are graphed in Figure 3-8. From the graph we can see that

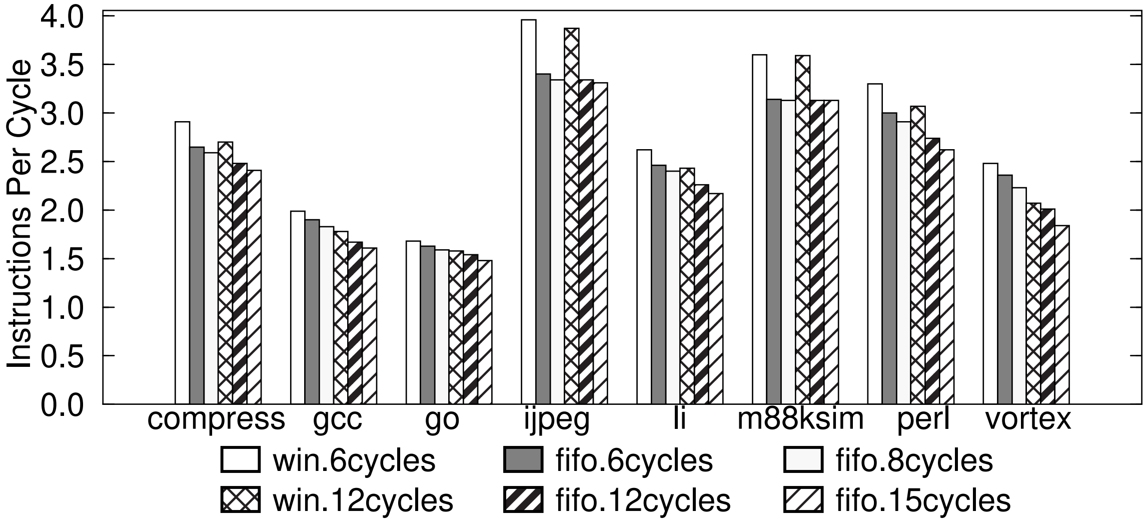


Figure 3-9. Effect of Scaling Instruction and Data Cache Miss Latency.

the dependence-based microarchitecture is capable of providing superior overall performance. The performance improvements vary from 9% to 21% with an average improvement of 14%.

Overall, our results show that the dependence-based microarchitecture using fifos is capable of superior performance due to its ability to support a fast clock while extracting significant levels of instruction-level parallelism.

3.2.5 Effect of Scaling Instruction and Data Cache Miss Latency

The clock advantage of the fifo-based microarchitecture could potentially increase cache miss latencies (measured in clock cycles). In order to quantify this effect, we studied the performance of the fifo-based microarchitecture when the cache miss latency is scaled by the same amount as the clock speed improvement. For example, a cache miss that took 6 cycles to complete would now take 8 cycles (7.5 cycles to be precise) due to the 25% improvement in clock speed.

Figure 3-9 graphs the results for base cache miss latencies of 6 cycles and 12 cycles. These latencies translate to 8 and 15 cycles respectively when the 25% clock speed advantage is taken into account. The "win.Ncycles" bars show the IPC for the window-based superscalar with same-cycle bypassing between functional units assuming a cache miss latency of N cycles. The "fifos.Ncycles" bars show the IPC for the 2-cluster fifo-based microarchitecture assuming a cache miss latency of N cycles. From the graph, we can see that the increase in cache miss latency due to clock speed improvement does not significantly impact the performance of the fifo-based microarchitecture. The highest reduction in IPC occurs for *gcc* — the performance reduction with respect to window-based superscalar went up from 4.5% to 8.0% when the cache miss latency is increased from 6 cycles to 8 cycles. The performance reductions are slightly higher when the base cache miss latency is increased to 12 cycles. The primary reason why the IPCs achieved for both the fifo-based microarchitecture and the window-based microarchitecture are not very sensitive to the cache miss latency for most benchmarks is the low cache miss rates of the benchmarks. The 32KB, 2-way L1 instruction and data caches are able to satisfy most of the memory accesses.

3.3 Other Dependence-based Microarchitectures

The microarchitecture presented in the previous section is one point in the design space of dependence-based microarchitectures. The fifo-based microarchitecture simplifies both the window logic and naturally reduces the performance degradation due to slow inter-cluster bypass paths. This section describes some other interesting points in the design space. In each case there are multiple clusters with inter-cluster bypasses taking multiple cycles to complete.

3.3.1 Single Window, Multiple Execution Clusters, Execution-driven Steering

In this design, shown in Figure 3-10, instructions reside in a central window while waiting for their operands and functional units to become available. Instructions are assigned to the clusters at the time they begin execution; this is *execution-driven* steering. With this steering, cluster assignment works as follows. The register values in the clusters become available at slightly different times, that is, the result register value produced by a cluster is available in that cluster one cycle earlier than in the other cluster. Consequently, an

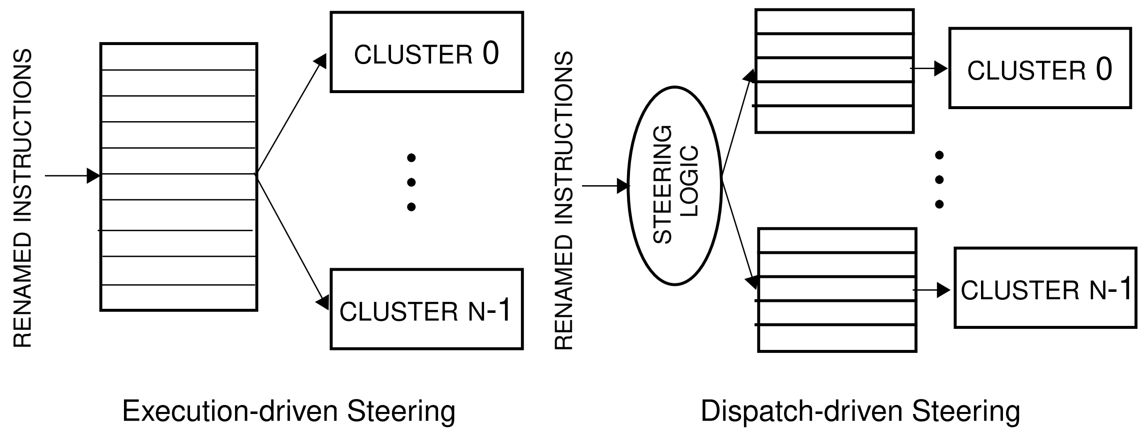


Figure 3-10. Other dependence-based microarchitectures.

instruction waiting for the value may be enabled for execution a few cycles (equal to the inter-cluster latency) earlier than in the other clusters. The selection logic monitors the instructions in the window and attempts to assign them to the cluster which provides their source values first (assuming there is a free functional unit in the cluster). Instructions that have their source operands available in all clusters are considered for assignment in a round-robin fashion starting with cluster 0. Static instruction order is used to break ties in this case.

The execution-driven approach uses a greedy policy to minimize the use of slow inter-cluster bypasses while maintaining a high utilization of the functional units. It does so by postponing the assignment of ready instructions to clusters until execution time. While this greedy approach might gain some IPC advantages, this design suffers from the previously discussed drawbacks of a central window and complex selection logic.

3.3.2 Multiple windows, Dispatch-driven Steering

This design, shown in Figure 3-10, is identical to the fifo-based microarchitecture presented in Section 3.2 except that each cluster has a completely flexible window instead of fifos. Instructions are steered to the windows using a heuristic that takes both dependences between instructions and the relative load of the clusters into account.

Steering Policies

In the case of dependence-based superscalar microarchitectures based on multiple windows with dispatch steering, we tried a number of steering heuristics. Three of these are described next.

1. *Fifo steering*. In this scheme the window is modeled as if it is a collection of fifos with instructions capable of issuing from any slot within each individual fifo. The fifos are only a conceptual device used by the instruction assignment heuristic — in reality, instructions issue from the window with complete flexibility. Instructions are steered to the "fifos" using the heuristic presented in Section 3.2. For example, a 32-entry window can be treated as eight fifos with four slots each. An advantage of considering the windows as a collection of fifos is that it helps to keep majority of the communication local and to achieve a good load balance at the same time.

2. *Round-robin steering*. In this scheme instructions in the dynamic stream are steered to clusters in a round-robin fashion with a particular block size. For example, for a block size of 16, the first 16 instructions are steered to cluster 0, the next 16 instructions are steered to cluster 1, and so on. The tacit assumption here is that dependences are localized in the dynamic stream as shown by previous studies on the distribution of ILP in programs [LW92,AS92]. In other words, instructions are dependent on other instructions that occur in close proximity (earlier) in the dynamic stream, i.e. independent instructions are well separated in the dynamic stream. An important parameter in this scheme is the block size. Using too small a block size can result in significant cross-cluster communication that can easily degrade performance by stretching the critical path. On the other hand using too big a block size can also degrade performance because now the number of functional units executing each block is a fraction of the total machine resources, i.e. low utilization might hurt performance. A compiler can assist this scheme by placing dependent instructions together. Studying the impact of instruction reordering by the compiler on the performance of this scheme is beyond the scope of this thesis.

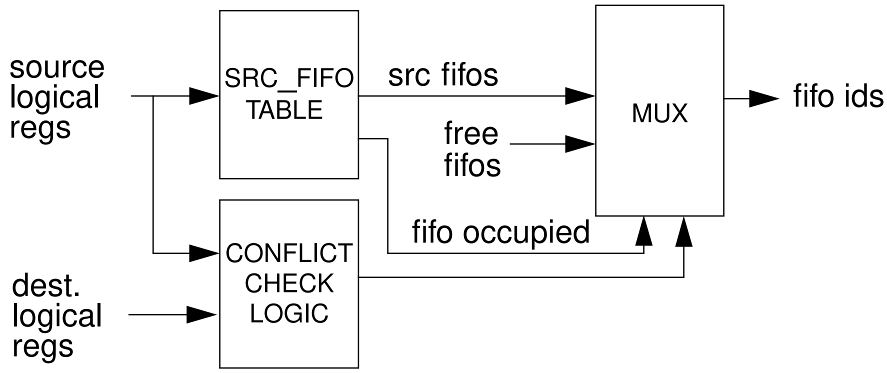


Figure 3-11. Fifo steering hardware.

3. *Random steering*. This steering heuristic is used as a basis for comparisons. Instructions are steered randomly to one of the clusters. If the window for the selected cluster is full, then the instruction is inserted into the other clusters in a round-robin fashion. This design point was evaluated in order to determine the degree to which dependence-based microarchitectures are capable of tolerating the extra latency introduced by slow inter-cluster bypasses and the importance of dependence-aware scheduling.

3.3.3 Complexity of Steering Policies

In addition to reducing inter-cluster communication and utilizing as many clusters as possible, a good steering policy must also be fast. Low latency is essential since any extra stages introduced in the front-end for steering can degrade performance (in terms of IPC) due to increased branch mispredict and instruction cache miss penalties. This can even nullify any advantages resulting from a faster clock. This section discusses the complexity of the steering policies analyzed in this chapter.

- *Fifo steering*. This steering policy can be implemented as shown in Figure 3-11. The logic operates in parallel with the register rename logic. The number of entries in the SRC_FIFO table is equal to the number of logical registers. The number of read ports and write ports into the SRC_FIFO table is $2 \times IW$ and IW respectively, where IW is the issue width. Comparing the block diagram with the one for rename logic, shown in Figure 2-3 on page 26, shows that the steering logic is functionally similar to the rename logic. There are two differences. First, the SRC_FIFO table is smaller than the rename map table as the width of each entry (determined by the number of fifos) is smaller than the width of the rename table. The second difference is that the output MUX in the case of fifo steering is slightly more complicated than that for the rename logic. Overall, the hardware complexity of fifo steering is similar to rename logic complexity. Just as shown for rename logic in Chapter 2, the delay of the steering logic increases linearly with issue width. Therefore, almost always the fifo steering logic can be performed in parallel with renaming. In the worst case, it might require an extra pipestage in addition to the rename stages.

- *Round-robin steering*. Since this simply requires a counter to count *block size* number of instructions before incrementing the "current" cluster pointer, the logic for steering is straightforward and can be accomplished in less time than the rename logic delay. Hence, steering in this case can be completely hidden behind renaming. Also, the delay of the steering logic is independent of issue width.
- *Random steering*. Just like in the case of round-robin steering, the logic required for random steering is straightforward and can be accomplished in less time than the rename logic delay. Hence, once again, steering can be completely hidden behind renaming. The delay of the steering logic is independent of issue width.

A natural question that arises in connection with instruction steering is: why cannot the compiler steer instructions? This question is especially pertinent given that the compiler has complete knowledge of register dependences between instructions and this is the critical information being used by the hardware to steer instructions. The key factor that makes the compiler less effective than hardware is the inability of the compiler to look beyond branches, i.e. detect the dynamic sequence of dependences created at run-time. Also, it is not obvious how the compiler can pass dependence information to the underlying hardware without compromising binary compatibility.

3.4 Experimental Evaluation

This section evaluates the performance of various dependence-based superscalar microarchitectures by measuring the performance of benchmark programs running on a detailed timing simulator. The timing simulator, a modified version of SimpleScalar [BAB96], is detailed in Table 3.2. All the configurations studied in this section are 8-wide — the configurations can fetch, decode, rename, and execute a maximum of eight instructions every cycle. An aggressive fetch mechanism is used to stress the issue and execution subsystems. The benchmark programs are from the SPEC'95 suite using their training input datsets. Each program was run for a maximum of 0.5B instructions

Fetch width	any 8 instructions
I-Cache	Perfect instruction cache
Branch predictor	McFarling's gshare [McF93] 4K 2-bit counters, 12 bit history unconditional control instructions predicted correctly
Issue window size	64
Maximum in-flight instructions	120
Retire width	16
Functional units	8 symmetrical units
Functional unit latency	1 cycle
Issue mechanism	out-of-order issue of up to 8 ops/cycle loads may execute when all prior store addresses are known
Physical registers	120int/120fp
D-Cache	32KB, 2-way SA write-back, write-allocate 32 byte lines, 1 cycle hit, 6 cycle miss four load/store ports

Table 3.2: Baseline simulation model

Simulated microarchitectures

Table 3.3 lists the various types of microarchitectures simulated here. The typical window-based microarchitecture, shown as the "1-cluster.1window" configuration, assumes uniform bypassing between all functional units within a single cycle, i.e. dependent instructions can execute back-to-back. All the dependence-based microarchitectures comprise two clusters with inter-cluster bypasses taking an extra cycle. The "2-cluster.1window.execsteer" configuration is made up of two execution clusters each containing half the execution resources of the machine. Renamed instructions are buffered in a central window and routed to the execution clusters using the execution-driven steering policy described in Section 3.3.1. In the "2-cluster.windows.randomsteer", "2-cluster.windows.fifosteer", and "2-cluster.windows.roundrobinsteer" configurations, both the window and execution resources are partitioned into two clusters and renamed instructions are routed to the clusters using random steering, fifo steering, and round-robin steering policies respectively. The "2-cluster.windows.randomsteer" design point was evaluated to determine the importance of dependence-aware scheduling. The "2-cluster.fifos.fifosteer" configuration is identical to the "2-cluster.windows.fifosteer" except that fifos are used in each cluster instead of a completely flexible window. Table 3.3 summarizes the various microarchitectures simulated.

Configuration	Window Organization	Steering Heuristic
window.execsteer	Flexible window	Execution steering
fifos.fifosteer	Fifos	Fifo steering
windows.fifosteer	Flexible window	Fifo steering
windows.roundrobinsteer	Flexible window	Round-robin steering
windows.randomsteer	Flexible window	Random steering

Table 3.3: Various microarchitectures simulated.

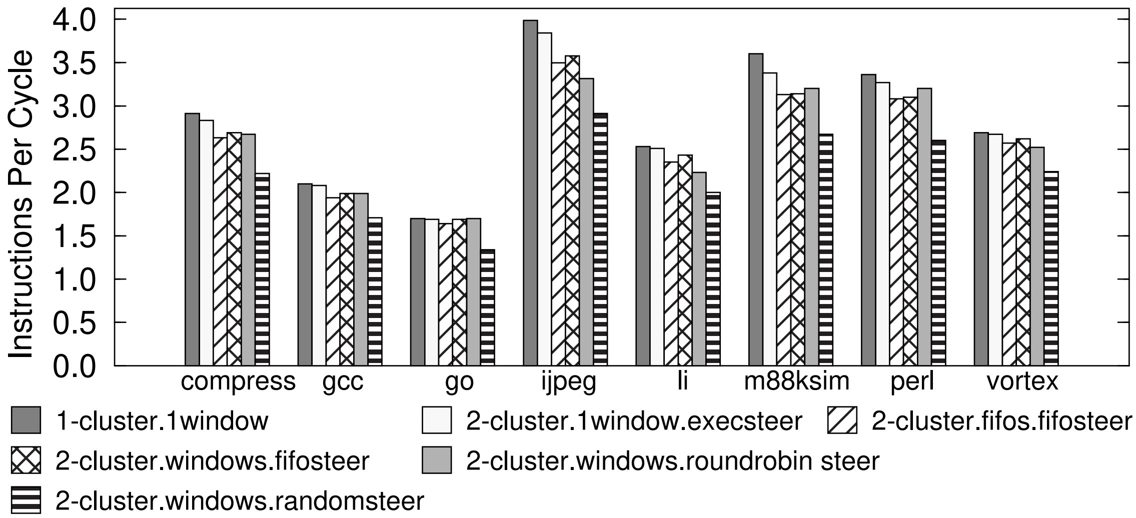


Figure 3-12. Performance of dependence-based superscalar microarchitectures.

3.4.1 Performance Relative to an Ideal Superscalar

The first set of experimental results, graphed in Figure 3-12, shows the performance of various dependence-based superscalar microarchitectures relative to a typical window-based microarchitecture in terms of instructions committed per cycle. A number of observations can be made from the graph. First, random steering consistently performs worse than the other schemes. The performance degradation with respect to the ideal case varies from 17% in the case of *vortex* to 23% in the case of *m88ksim*. Hence, it is essential for the steering logic to consider dependences when routing instructions. Second, the microarchitecture with a central window and execution steering performs nearly as well as the ideal microarchitecture with a maximum degradation of 3% in the case of *m88ksim*. However, as discussed earlier in Section 3.3.1, this microarchitecture requires a centralized window with complex selection logic. Third, the "2-cluster.fifos.fifosteer", "2-cluster.windows.fifosteer", and "2-cluster.windows.roundrobin steer" microarchitectures perform competitively in comparison to the ideal microarchitecture. As expected, using completely flexible windows instead of fifos helps improve performance slightly. Another way of interpreting this result is that it reinforces the earlier finding that windows can be replaced with the combination of fifos and intelligent steering with little degradation in IPC. An

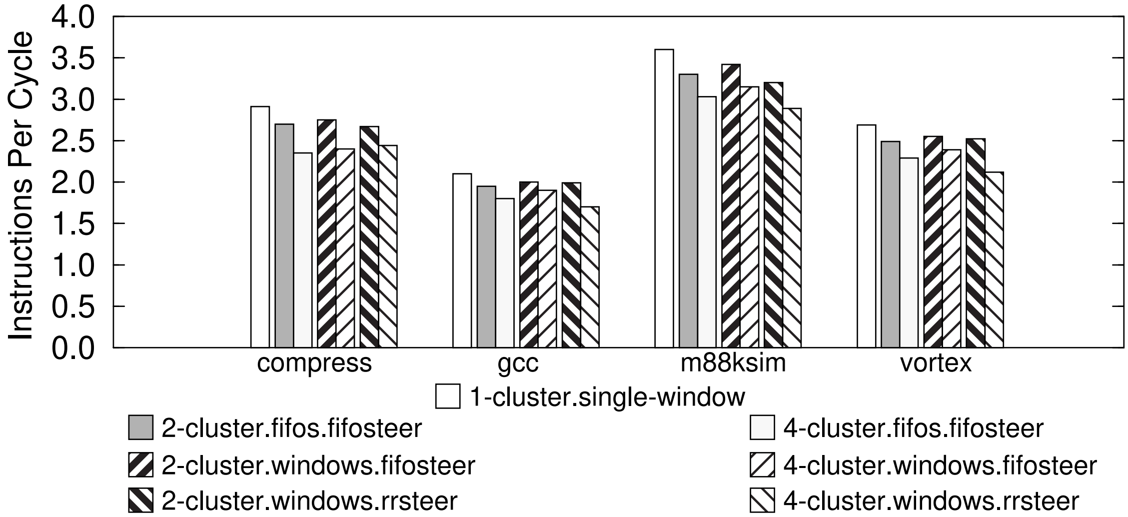


Figure 3-13. Effect of increasing number of clusters.

interesting supplementary result is that round-robin steering, which can be implemented using simple logic, performs as well as the more complex fifo steering. However, as shown later, round-robin steering does not scale well as the number of clusters and is increased.

Overall, the above results show that dependence-based superscalar microarchitectures can deliver performance similar, in terms of instructions committed per cycle, to that of an ideal microarchitecture with a large window and uniform, single cycle bypasses between all functional units.

3.4.2 Effect of Increasing Number of Clusters

The graph in Figure 3-13 shows the effect of increasing the number of clusters on the performance of “fifos.fifosteer”, “windows.fifosteer”, and the “windows.rrsteer” microarchitectures. Performance uniformly degrades for the three designs as the number of clusters is increased. This is expected since increasing the number of clusters augments load imbalance and results in more frequent inter-cluster communication. The performance degradation going from 2 clusters to 4 clusters for the “fifos.fifosteer” and “windows.fifosteer” microarchitectures is in the 5%-10% range. For the “windows.rrsteer” microarchitectures the performance degradation is in the 9%-17% range. For all the benchmarks, the performance of the round-robin steering policy degrades more than the fifo steering policy. This is mainly due to two reasons. First, the fifo steering policy does a better job of exploiting the full width of the machine. For example, it can use all the clusters cooperatively to execute a block of instructions. In the case of round-robin steering, the block of instructions might be steered to a single cluster and hence, only the resources in that cluster can be employed to execute the instructions, resulting in lower throughput. The second reason for the superior performance of the fifo steering policy is that it requires fewer inter-cluster bypasses as compared to the round-robin steering heuristic. A simple example explains this. Consider the case where there are 4 clusters each 2-wide (2 functional units) and the dynamic stream is made up of two chains (parallelism is equal to 2). In this situation, the fifo steering policy will only utilize a single cluster since all instructions will be routed to the two fifos in the cluster. This eliminates inter-cluster communication completely in this example. The round-robin steering policy on the other hand, is oblivious of the parallelism in the instruction stream, and uniformly steers instructions to all available clusters. Therefore, in this case, inter-cluster communication is more frequent with the round-robin steering policy than with the fifo steering policy.

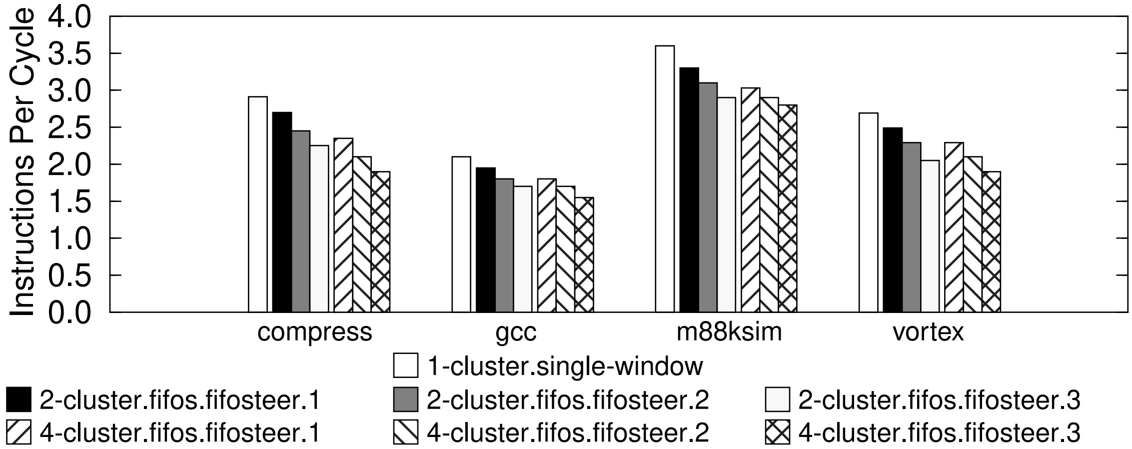


Figure 3-14. Effect of increasing inter-cluster latency.

3.4.3 Effect of Increasing Inter-cluster Latency

The graph in Figure 3-14 shows the effect of increasing inter-cluster latency on the performance of 2-cluster and 4-cluster "fifos.fifosteer" microarchitectures. Performance degrades as the latency of inter-cluster communication is increased. This is expected since increasing inter-cluster communication latency increases the time taken to perform any computation that is spread across multiple clusters and hence, could easily stretch the critical path of the program. For 2-cluster configurations, the average performance degradation for 2-cluster systems when the inter-cluster latency is increased from 1[1] to 2 and from 2 to 3 cycles is 8.7% and 9.3% respectively. Similarly, for 4-cluster systems, the corresponding performance degradations are 13.4% and 11.2% respectively. The reduction in performance is higher for the 4-cluster systems since the number of instruction dependences spread across clusters increases with the number of clusters. This shows that it is extremely important to provide low latency inter-cluster communication for high performance.

1. There is a single bubble between two dependent instructions executing in different clusters.

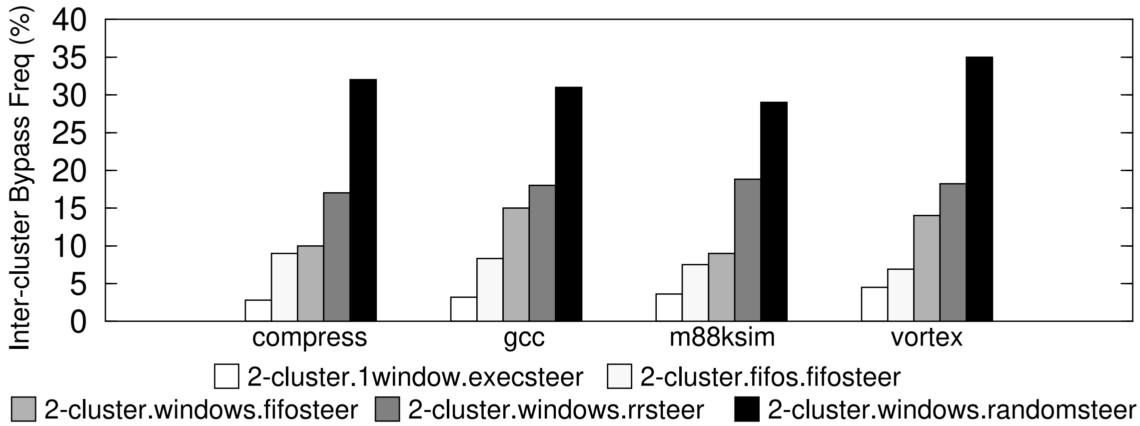


Figure 3-15. Inter-cluster bypass frequency.

3.4.4 Inter-cluster Bypass Frequency

The graph in Figure 3-15 shows the frequency of inter-cluster communication for various steering heuristics and 4-cluster configurations. Inter-cluster communication is measured in terms of the fraction of total instructions that exercise inter-cluster bypasses. This does not include cases where an instruction reads its operands from the register file in the cluster i.e. cases in which the operands arrive from the remote cluster in advance. As expected, we see that there is a high correlation between the frequency of inter-cluster communication and performance - configurations that exhibit higher inter-cluster communication commit fewer instructions per cycle. The inter-cluster communication is particularly high in the case of random steering, reaching as high as 35% in the case of *vortex*. Execution steering exhibits the lowest inter-cluster bypass frequency. This is not surprising because execution steering is based on the greedy policy of postponing selection to favor execution of dependent instructions in the same cluster. Another observation that can be made from the graph is that the “fifos.fifosteer” microarchitecture uniformly exercises fewer inter-cluster bypasses than the “windows.rrsteer” microarchitecture. This is in agreement with earlier discussion about how the fifo steering policy dynamically adapts to the number of clusters being used based on the parallelism in the instruction stream, thus resulting in fewer inter-cluster bypasses.

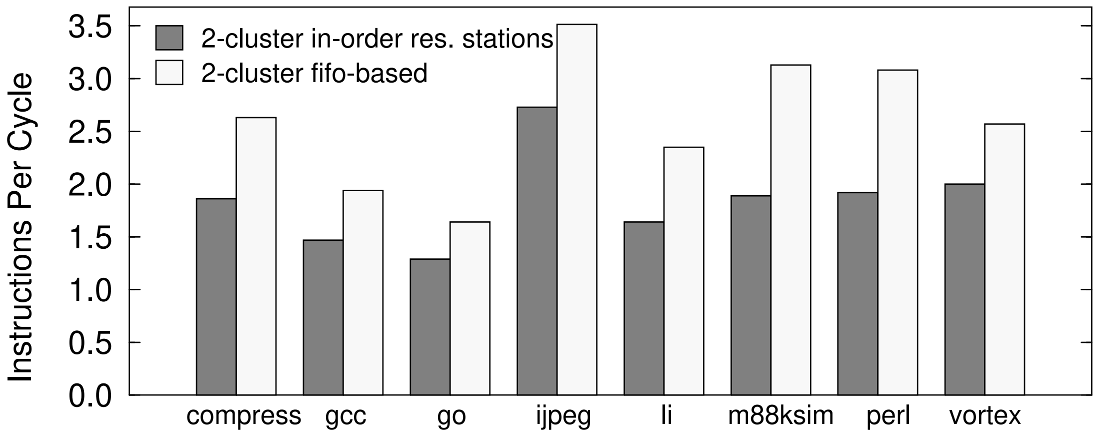


Figure 3-16. Comparing against in-order distributed reservation stations.

3.4.5 Comparing against In-order Distributed Reservation Stations

Johnson [Joh91] proposed using in-order distributed reservation stations as a means of reducing the complexity of the instruction window. Instructions are forced to issue in-order from the reservation stations. The advantages of such a scheme are similar to those of the fifo-based microarchitecture; simpler wakeup and selection logic. The fifo-based microarchitecture differs from Johnson's scheme in the manner in which instructions are steered to the fifos. The dependence-based microarchitecture steers instructions based on dependence information extracted at run-time instead of instruction type as in the case of the in-order reservation stations scheme.

The graph in Figure 3-16 compares the performance of 2-cluster configurations based on in-order distributed reservation stations and fifo-based microarchitecture with fifo steering policy respectively. The dependence-based microarchitecture consistently performs better than in-order reservation stations. The average performance degradation is as high as 27%. This is mainly due to two factors. First, in the in-order reservation stations scheme, instructions at the head of the reservation stations can block other ready instructions behind them from issuing. Second, the instruction distribution logic in the in-order reservation stations scheme makes no attempt to minimize the use of inter-cluster bypasses.

Butler and Patt [BP92] also report significant performance degradation when the "head-only" (fifo) scheduling policy is used with distributed reservation stations.

3.5 Related Work

Tomasulo, in his original proposal [Tom67] on dynamic scheduling, proposed distributed reservation stations as an alternative to centralized reservation stations to reduce complexity. Distributed reservation stations simplify selection logic. The selection logic at a functional unit only has to monitor the instructions in the reservation stations associated with that unit. However, the result tags still have to broadcast to all the reservation stations just as in the case of centralized reservation stations, i.e. the complexity of window wakeup logic remains the same.

Johnson [Joh91] proposed in-order distributed reservation stations to further reduce issue-logic complexity. The fifo-based microarchitecture presented in this chapter is similar to the in-order distributed reservation stations scheme in a number of respects. Both distribute window entries and force in-order issue out of the distributed window entries to simplify selection logic. However, there are two key differences. First, the fifo-based microarchitecture uses a *prescheduling* (steering) phase to determine a suitable fifo to place each instruction in. As shown in Section 3.4.5, this intelligent steering helps the dependence-based microarchitecture extract more parallelism relative to in-order distributed reservation stations. Second, the dependence-based microarchitectures use clustering to simplify wakeup logic. A cluster consists of a small number of branch , ALU, and memory units. Window operations and bypasses within a cluster complete within a single cycle, thus facilitating back-to-back execution of dependent instructions residing in each cluster. Tomasulo's distributed reservation stations on the other hand clusters functional units based on type. For example, all memory units are clustered together and so on. This results in more cross-cluster traffic compared to the dependence-based microarchitectures.

An early CRAY-2 design [Unk79,SS90,Smi97] realized the importance of detecting and exploiting dependences to facilitate a fast clock. The issue logic consisted of four instruction queues feeding eight execution units. A dependent chain of instructions were issued to the same queue. The compiler was responsible for grouping dependent instructions together. A single accumulator style instruction set helped express the grouping to the hardware without the need for extra bits to explicitly specify dependences. The hardware simply starts a new chain whenever it hits a LDA (*load accumulator)* instruction in the instruction stream. As a result, the hardware does not have to extract dependence information at run-time. The fifo-based microarchitecture investigated in this chapter was partly inspired by the CRAY-2 design. The primary difference is that hardware steering is used instead of compiler steering. As explained before, hardware steering is well-suited for integer codes since the small basic blocks and frequent control instructions in integer codes can severely handicap compile-time steering of instructions to fifos.

Kemp and Franklin [KF96] studied a microarchitecture called PEWS (Parallel Execution Windows) for simplifying the logic associated with a central window. PEWs simplifies window logic by splitting the central instruction window among multiple windows much like the dependence-based microarchitectures described in this chapter. Register values are communicated between clusters (called pews) via hardware queues and a ring interconnection network. In contrast, we assume a broadcast mechanism for the same purpose. Instructions are steered to the pews based on instruction dependences with a goal to minimize inter-pew communication. However, for their experiments Kemp and Franklin assume that each of the pews has as many functional units as the central window organization. This assumption implies that the reduction in complexity achieved is limited because the wakeup and selection logic of the windows in the individual pews still have the same porting requirements as the central window.

The DEC 21264 [Gwe96a] is the first commercial microarchitecture implementing out-of-order scheduling that was forced to use significant microarchitectural changes, relative to the conventional microarchitecture, to support a fast clock. Like the dependence-based microarchitectures explored in this chapter, the execution units are partitioned into two clusters with bypasses between clusters taking an extra cycle to complete. The selection logic steers instructions buffered in a central window to the execution cluster based on dependences. The exact steering algorithm used has not been made public.

Multiscalar processors [Bre,FS92, Fra93,SBV95] pioneered the concept of using decentralized processor resources to reduce complexity. Multiple clusters, each similar in structure to a narrow superscalar, are used to execute different portions of the serial program. The different portions of the program are called *tasks* and can be identified either by the compiler or by the hardware. The design is highly decentralized. All major structures in the pipeline, starting from the fetch hardware, are distributed. In addition, the paradigm naturally supports advanced features like *multiple flows of control* and *out-of-order fetch*. These features are considered essential for exploiting higher levels of parallelism [LW92, Smi95] in future. While the Multiscalar design is a futuristic microarchitecture designed with complexity-effectiveness in mind, it will take some time for the design to evolve and for its implementation to become feasible. The dependence-based superscalar microarchitectures explored in this chapter provide a smooth transition path, from the point of view of implementation, to Multiscalar-like designs from current superscalar designs.

More recently, processor microarchitectures called Trace processors [VM97, RJSS97] have been proposed that organize the microarchitecture around traces. Just like in the Multiscalar and dependence-based microarchitectures, execution resources are partitioned into clusters. Each cluster is assigned a dynamic instruction trace for execution that is fetched from a cache of traces called the trace cache. The trace cache in addition to providing a high-bandwidth fetch mechanism also simplifies rename logic by caching rename information along with the trace. The trace processor microarchitecture can be viewed as a dependence-based microarchitecture that has completely flexible windows in each cluster and steers instructions to clusters using a round-robin policy.

Farkas et al. [FCJV97] propose the multicluster microarchitecture to reduce the clock cycle time of typical superscalar microarchitectures. The multicluster microarchitecture is similar in concept to the dependence-based microarchitectures explored here. There are two primary differences, however. First, the multicluster architecture uses compiler steering instead of hardware steering. Second, it uses explicit copy instructions to communicate operand values between the clusters. Steering information is passed to the hardware indirectly without changing the instruction set architecture. Each cluster is assigned a subset of the architectural registers and instructions are steered based on the registers specified in the instruction. A static scheduling heuristic chooses a cluster so that the load imbalance between the two clusters[1] is minimized. Farkas et al. found that even this heuristic cannot be directly addressed by the compiler because the work done by a cluster is a function of the order in which instructions are issued, and the issue order is not deterministic for dynamically-scheduled processors.

3.6 Chapter Summary

This chapter presented the design and evaluation of a family of complexity-effective microarchitectures called dependence-based superscalar microarchitectures. These microarchitectures facilitate a fast clock while exploiting similar levels of parallelism as an ideal large-window machine. The proposed microarchitectures use a two-pronged strategy for high performance. First, the issue window and execution resources are partitioned to facilitate a fast clock. Second, instructions are intelligently steered, taking into account dependences, to the different partitions in order to extract similar levels of parallelism as an ideal large-window machine.

One of the dependence-based microarchitectures, called the fifo-based microarchitecture, detects chains of dependent instructions and steers the chains to fifos which are constrained to execute in-order. Since only the instructions at the fifo heads have to be monitored for execution, the proposed microarchitecture simplifies window logic. Furthermore, the microarchitecture naturally lends itself to clustering by grouping dependent instructions together. This grouping of dependent instructions helps mitigate the bypass problem to a large extent by using fast local bypasses more frequently than slow inter-

1. They only study 2-cluster systems.

cluster bypasses. The performance of a 2 X 4-way fifo-based microarchitecture is compared with a typical 8-way superscalar. The results show two things. First, the proposed microarchitecture has IPC performance close to that of a typical microarchitecture (average degradation in IPC performance is 7.8%). Second, when taking the clock speed advantage of the fifo-based microarchitecture into account the 8-way proposed microarchitecture is 14% faster than the typical window-based microarchitecture on average.

Overall, the experimental results presented show that dependence-based superscalar microarchitectures are capable of extracting similar levels of parallelism as typical microarchitectures while enabling a faster clock.

Chapter 4

Integer-Decoupled Microarchitecture

The integer-decoupled microarchitecture is a complexity-effective microarchitecture that can improve the performance of integer programs with little or no increase in complexity. It is particularly attractive since it can be implemented on top of current microarchitectures with relatively small hardware changes. This chapter proposes and evaluates the integer-decoupled microarchitecture.

Integer-decoupled microarchitectures execute some of the integer instructions, those not involved in computing addresses and accessing memory, on idle floating-point resources that have been augmented to perform simple integers operations. The compiler identifies computation to *off-load* to the floating-point subsystem. This results in a number of benefits for integer programs including extra issue width, a bigger effective window, and decoupling of memory access from the actual computation.

Another way to look at the integer-decoupled microarchitecture, in the context of dependence-based microarchitectures presented in previous chapter, is that the existing floating-point subsystem provides an extra cluster, for free, that can used for executing integer instructions. However, unlike the dependence-based microarchitectures, instruction steering in this case is performed by the compiler.

The rest of the chapter is organized as follows. Section 4.1 presents the concept behind the integer-decoupled microarchitecture. Section 4.2 discusses the hardware additions that have to be made to the conventional microarchitecture. Section 4.3 illustrates, with an example, the kind of computation that is off-loaded to the augmented FP subsystem. Section 4.4 discusses the role of the compiler and the basic partitioning scheme used by the compiler. Section 4.5 shows how the basic partitioning scheme can be improved using copy instructions and code duplication. Section 4.6 presents the results of an experimental evaluation of the proposed microarchitecture. Finally, the chapter is summarized in Section 4.8.

4.1 Concept

To motivate the proposed microarchitecture, consider how the conventional microarchitecture illustrated in Figure 1-1 on page 2 works. The instruction fetch unit reads multiple instructions from the instruction cache and feeds them to integer and floating-point subsystems for execution. The integer subsystem contains a number of load/store, branch, and functional units that operate on integer operands. The floating-point subsystem is similar to the integer subsystem except it does not contain load/store units, and it operates on floating-point operands. Instruction windows, in the form of buffers, are used to decouple the instruction fetch unit from the integer and floating-point execution subsystems.

Partitioning issue and execution resources into integer and floating-point subsystems has several advantages. First, as shown in Chapter 2, it eliminates the cycle time penalties associated with centralized structures. For example, registers are divided into integer and floating-point files, each with a set of ports. And, the instruction window is similarly divided with separate issue logic. Second, while executing floating-point programs, the microarchitecture naturally decouples addressing and floating-point computation: address computation executes in the integer subsystem while floating-point computation executes in the FP subsystem so that dynamic scheduling between the two can be enhanced. Third, since integer data and floating-point data typically have different widths (32-bit versus 64-bit), using separate integer and floating-point subsystems helps reduce implementation complexity and save silicon area. The last benefit will be nullified by the move towards 64-bit instruction set architectures in which both integer and floating-point data are 64 bits wide. The uniform use of 64-bit data in both integer and floating-point subsystems enables the optimization being proposed here.

This microarchitecture style leads to idle floating-point resources — registers, functional units, instruction window logic, and buses — while executing integer programs or integer-intensive portions of floating-point programs. To address this drawback, we propose a more general decoupled microarchitecture style based on earlier work [BRT93,GHL+85,PD83,Smi82,S+87], in which the floating-point subsystem executes both integer and floating-point operations. In this microarchitecture, which we refer to as the *integer-decoupled* microarchitecture, a load/store subsystem (LdSt) that mostly executes integer instructions involved in effective address calculation and memory access. A computation (Comp) subsystem supports all floating-point operations as well as non-memory related integer computation. The integer decoupled microarchitecture can be built on top of the conventional microarchitecture with relatively few hardware additions. These hardware changes are discussed in the next section.

The integer-decoupled microarchitecture has a number of performance advantages over a conventional microarchitecture for integer programs. First, it provides extra issue and execution bandwidth for integer programs. For example, by implementing the integer-decoupled microarchitecture, a superscalar processor with 2 integer and 2 floating-point functional units can provide an issue and execution width of 4 for most integer codes. Second, by using the instruction window in the floating-point subsystem, the integer-decoupled microarchitecture provides a larger overall window. This can potentially increase the amount of parallelism exploited. Third, the compiler now has 64 logical registers (32 int and 32 fp) for holding integer variables instead of the usual 32. Finally, the integer-decoupled microarchitecture often facilitates early resolution of mispredicted branches. If the branch computation associated with a mispredicted branch executes in the less heavily loaded Comp subsystem then it is very likely that the branch will be resolved earlier relative to the conventional microarchitecture

The integer-decoupled concept can also be used to reduce the complexity of a conventional superscalar microarchitecture. By steering integer instructions to the augmented floating-point subsystem, the integer-decoupled microarchitecture does not require as many issue window entries in the integer subsystem as the conventional microarchitecture. Similarly, it can be used to reduce the size of the physical register file in the integer subsystem. Ideally, the complexity of a n-wide conventional microarchitecture can be reduced by implementing it as an integer-decoupled microarchitecture with the LdSt and Comp subsystems each being $n/2$-wide. This advantage of the integer-decoupled microarchitecture is not quantified here.

4.2 Changes to the Conventional Microarchitecture

The integer-decoupled microarchitecture remains very similar to a conventional microarchitecture. The only hardware modification required is augmenting the existing floating-point functional units to perform simple integer operations. There needs to be no additional cost for registers and buses if the integer operations are embedded in the existing floating-point functional units and share the existing register file ports and buses. Similarly, instruction fetch and issue resources are unchanged. The only extra costs are the additional gates required to implement the simple integer operations and the opcodes for specifying these operations. Results presented later show that the gate-intensive integer multiply and divide operations need not be duplicated and hence, the extra cost should not be a factor.

The instruction set architecture (ISA) has to be minimally augmented to include the simple integer operations that operate on the floating-point registers. The changes required are similar in spirit to the recent multimedia extensions introduced by most microprocessor vendors [Gwe95c, Gwe96b]. The integer opcodes of the SimpleScalar [BAB96] ISA that are supported in the Comp subsystem are shown in Table 4.1. Because the floating-point opcode space is usually relatively sparse compared to the integer opcode space, and about 21 extra opcodes are required, the necessary ISA extensions are realistic.

Operation type	Opcodes
Control	bgez bgtz blez bltz bne
Logical	andi nor ori xori sllv sll srav sra srlv srl
Arithmetic	addi addiu addu lui slti sltiu

Table 4.1: Extra opcodes supported in the Comp subsystem.

4.3 Partitioning the Program

Given the constraints of the integer-decoupled microarchitecture, let us look at the kind of integer computation that can be off-loaded to the Comp subsystem and the role of the compiler in identifying such computation. Because we want to decouple address computation from the rest of the program computation, all load/store instructions and integer instructions involved in effective address computation are assigned to the LdSt subsystem. All other sequences of instructions terminate either in the computation of branch outcomes or store values. The instruction sequences, called branch computation and store-value computation, are ideal candidates for execution in the Comp subsystem because they do not require any special support in the Comp subsystem. The result of a branch computation, the branch outcome, is sent to the fetch unit where it is used to validate the predicted outcome. This functionality is present in existing floating-point subsystems for floating-point branches. The result of a store-value computation, the value being stored, is deposited in the write buffer where it merges with the corresponding store address generated by the LdSt subsystem. This mechanism is also implemented in current floating-point subsystems to store floating-point values. However, some store-value and branch computations might not be assigned to the Comp subsystem if the instructions in these computa-

```
extern unsigned long regs_inv_by_call;

for (regno = 0; regno < FIRST_PSEUDO_REG; regno++)
    if (regs_inv_by_call & (1 << regno)) {
      delete_equiv_reg(regno);
      if (reg_tick[regno] >= 0)
         reg_tick[regno]++;
    }
```

```
I1:         move     $16, $0                                      /* regno = 0 */
I2:  $L5:   lw       $2, regs_inv_by_call
I3:         sra      $2, $2, $16
I4:         andi     $2, $2, 0x1        /* $2=regs_inv_by_call & (1<<regno) */
I5:         beq      $2, $0, $L4
I6:         move     $4, $16
I7:         jal      delete_equiv_reg
I8:         lw       $3, reg_tick
I9:         sll      $2, $16, 2
I10:        addu     $2, $2, $3
I11:        lw       $4, 0($2)                        /* $4 = reg_tick[regno] */
I12:        bltz     $4, $L4
I13:        addu     $4, $4, 1
I14:        sw       $4, 0($2)                           /* reg_tick[regno]++ */
I15: $L4:   addu     $16, $16, 1                                 /* regno++ */
I16:        slt      $2, $16, 66
I17:        bne      $2,$0,$L5           /* regno < FIRST_PSEUDO_REG */
```

Figure 4-1. An example program fragment.

tions are also involved in address computation. The example to be presented next illustrates this.

Figure 4-1 shows a program fragment in C from invalidate_for_call, a frequently executed function in the SPEC benchmark *gcc*. The for-loop in the program runs through all the pseudo registers and does some bookkeeping for those that are invalidated by function calls. The figure shows assembly code compiled for a conventional microarchitecture. The whole program executes in the integer subsystem leaving the floating-point subsystem completely idle.

With very little effort, the assembly code shown in Figure 4-1 can be transformed to offload some of the integer computation to the Comp subsystem as shown on the left in

Basic partitioning scheme

I1:		move	$16, $0
I2:	$L5:	lw	$2, regs_inv_by_call
I3:		sra	$2, $2, $16
I4:		andi	$2, $2, 0x1
I5:		beq	$2, $0, $L4
I6:		move	$4, $16
I7:		jal	delete_equiv_reg
I8:		lw	$3, reg_tick
I9:		sll	$2, $16, 2
I10:		addu	$2, $2, $3
I11:		*lw*	*$f0, 0($2)*
I12:		**bltz,c**	**$f0, $L4**
I13:		**addu,c**	**$f0, $f0, 1**
I14:		*sw*	*$f0, 0($2)*
I15:	$L4:	addu	$16, $16, 1
I16:		slt	$2, $16, 66
I17:		bne	$2,$0,$L5

Basic partitioning scheme

I1:		move	$16, $0
I1':		*cp_comp*	*$f2, $16*
I2:	$L5:	*lw*	*$f4, regs_inv_by_call*
I3:		**sra,c**	**$f4, $f4, $f2**
I4:		**andi,c**	**$f4, $f4, 0x1**
I5:		**beq,c**	**$f4, $0, $L4**
I6:		move	$4, $16
I7:		jal	delete_equiv_reg
I8:		lw	$3, reg_tick
I9:		sll	$2, $16, 2
I10:		addu	$2, $2, $3
I11:		*lw*	*$f0, 0($2)*
I12:		**bltz,c**	**$f0, $L4**
I13:		**addu,c**	**$f0, $f0, 1**
I14:		*sw*	*$f0, 0($2)*
I15:	$L4:	addu	$16, $16, 1
I15':		**addu,c**	**$f2, $f2, 1**
I16:		**slt,c**	**$f4, $f2, 66**
I17:		**bne,c**	**$f4,$0,$L5**

Advanced partitioning scheme

Figure 4-2. Code partitioning for example fragment.

Figure 4-2. Integer instructions that execute in Comp are shown in **bold** with a **,c** suffix. The load instruction, I11, instead of loading into integer register $4, now loads the value into floating-point register $f0. Instructions I12 and I13 operate on the loaded value in floating-point register $f0 and execute in the Comp subsystem. The result of the branch instruction (I12) is sent from the Comp subsystem to the fetch unit to validate the prediction made. The result of the add instruction (I13) is sent to the store buffer where it is merged with the address generated by the store instruction (I14) executed in the LdSt subsystem. The load and store instructions (I11 and I14) are italicized to point out that these instructions now load and store floating-point registers. These are the same as floating-point load and store instructions in the conventional microarchitecture. Relating the example to the discussion earlier, the branch computation and store-value computation that are off-loaded in this case are the singleton sets {I12} and {I13} respectively. The branch computation {I15, I16, and I17} was not assigned to the Comp subsystem because instruction I15 is also involved in generating the address for the load instruction I11.

In the transformation just presented, computation was off-loaded to the Comp subsystem without introducing new instructions in the program. However, by strategically inserting *copy* instructions and duplicating some instructions, additional computation can be off-loaded to the Comp subsystem. For example, consider the transformation presented on the right in Figure 4-2. The copy instruction (I1') and the duplicate instruction (I15') help off-load a sizable fraction of the total computation to the Comp subsystem. Now, as many as seven static instructions of the original program execute in the Comp subsystem.

The compiler for the integer-decoupled microarchitecture is responsible for effecting the transformations presented above. More abstractly, the compiler is responsible for partitioning the original program into LdSt and Comp partitions. The transformation on the left in Figure 4-2 is a result of the *basic* partitioning scheme used by the compiler. In this scheme, no new instructions are introduced and communication between the two subsystems happens via loads and stores that already exist in the original program. Section 4.4 discusses the basic scheme in detail. The second transformation is a result of the advanced partitioning scheme used by the compiler. In this scheme, the compiler intelligently introduces a few extra instructions in the form of copy or duplicate instructions to enable off-loading of more computation to the Comp subsystem. Section 4.5 discusses the advanced partitioning scheme.

4.4 Basic Partitioning Scheme

As mentioned earlier, the basic partitioning scheme off-loads computation to the Comp subsystem without introducing new instructions. In this section, some terminology is presented first to aid subsequent discussion. Then, the necessary conditions that need to be satisfied for branch and store-value computation to be assigned to the Comp subsystem are described. Finally, the partitioning algorithm used by the compiler is presented.

Program P	Backward-Slice(P,f)	Forward-Slice(P,a)
a = b + c;	a = b + c;	
d = a * g;	d = a * g;	d = a * g;
f = d + 2;		f = d + 2;

Figure 4-3. Program slices.

4.4.1 Terminology and Data Structures

A slice [Wei84] of a program *P* with respect to a value *v* is defined to be the subset of *P* that is involved in the computation of *v*. We term this the *backward slice* of *P* with respect to *v* and represent it as *Backward-Slice(P,v)*. The forward slice of *P* with respect to *v* is all computation that is affected by *v*, and is represented as *Forward-Slice(P,v)*. An example is shown in Figure 4-3.

To partition a program, the compiler uses a data structure called the static dependence graph that compactly represents all the register dependences in a program. The static dependence graph (SDG) is a directed graph which has a node corresponding to each static instruction in the program. The SDG has an edge from node v_i to node v_j if instruction *i* produces a register value that *could* be consumed by instruction *j*. Load and store instructions are special cased in the SDG to simplify the partitioning algorithm. Each load instruction is split into two nodes - one representing the load address and the other representing the loaded value. Similarly, each store instruction is split into two nodes - one representing the store address and the other representing the store value. This is done because a load instruction executes in the LdSt subsystem, but the value can be loaded into either subsystem. Likewise, the value being stored can come from either the LdSt subsystem or the Comp subsystem.

Figure 4-4 shows the SDG for the program fragment in Figure 4-1. Nodes 2, 8, and 11 correspond to load instructions and have been split. To show that both nodes correspond to a single program instruction, the split nodes have been enclosed in a bigger oval node.

```
I1:          move   $16, $0
I2:  $L5:    lw     $2, regs_inv_by_call
I3:          sra    $2, $2, $16
I4:          andi   $2, $2, 0x1
I5:          beq    $2, $0, $L4
I6:          move   $4, $16
I7:          jal    delete_equiv_reg
I8:          lw     $3, reg_tick
I9:          sll    $2, $16, 2
I10:         addu   $2, $2, $3
I11:         lw     $4, 0($2)
I12:         bltz   $4, $L4
I13:         addu   $4, $4, 1
I14:         sw     $4, 0($2)
I15: $L4:    addu   $16, $16, 1
I16:         slt    $2, $16, 66
I17:         bne    $2,$0,$L5
```

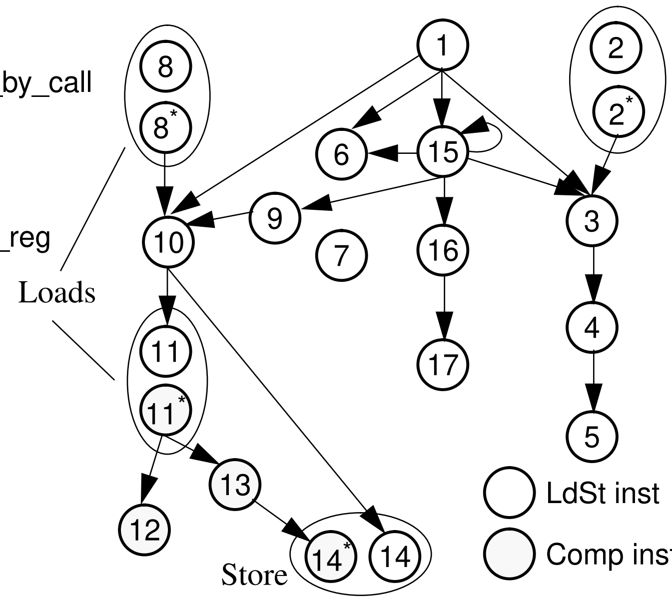


Figure 4-4. Static dependence graph for example program.

Similarly, node 14 corresponds to a store instruction and has been split. The edges correspond to register dependences. For example, instruction I3 produces \$2 that is used by instruction I4 and hence, there is an edge between I3 and I4.

4.4.2 Partitioning Conditions

Given a program *P*, let

G = SDG for *P*

LS(G) = Set of load/store address nodes in *G*

C(G) = Comp partition of *G*

L(G) = LdSt partition of *G*

Any partition of *G* into *L(G)* and *C(G)* must satisfy two conditions. First, *L(G)* and *C(G)* must be disjoint. Second, a node $v \in C(G)$ should satisfy the following conditions:

1. $Backward\text{-}Slice(G,v) \cap L(G) = \emptyset$. For a node $v \in C(G)$, this conditions specifies that v or any of its ancestors should not receive any value from *L(G)*.

2. *Forward-Slice(G,v)* $\cap$ *L(G)* = Ø. For a node $v \in C(G)$, this condition specifies that v or any of its descendants should not supply any value to *L(G)*.

Clearly, nodes in *LS(G)* must be in *L(G)* because only the LdSt subsystem can execute loads and stores. Instructions in the backward slices of these address nodes are involved in addressing. The union of these backward slices is termed the LdSt slice. It follows from the backward slice condition that the LdSt slice must also be assigned to *L(G)*.

For our example program repeated in Figure 4-4,

LS(G) = {2, 8, 11, 14},

C(G) = {11^*, 12, 13, 14^*}, and

L(G) = *G* - *C(G)* = {1, 2, 2^*, 3, 4, 5, 6, 7, 8, 8^*, 9, 10, 11, 14, 15, 16, 17}

It can be easily verified that all nodes in *C(G)* satisfy the backward and forward slicing conditions. The branch computation {16, 17} could not be assigned to the Comp subsystem because node 16 is supplied a value by node 15 which is in the LdSt slice and hence in *L(G)*. If this branch computation were assigned to Comp, then the backward slice condition would be violated for nodes 16 and 17.

4.4.3 Partitioning Algorithm

The goal of the partitioning algorithm is to find the largest set *C(G)* that satisfies the partitioning conditions presented previously. A simple and fast algorithm for identifying the largest set *C(G)* based on the observation that the partitioning conditions specified previously can be restated as reachability conditions on the undirected graph G_u corresponding to G.

Let G_u be the undirected graph corresponding to G, i.e. G_u consists of the same vertices and edges as G, but the edges are undirected. Then, the slicing conditions can be interpreted as : If $v \in C(G)$, then v is not reachable from any node in $L(G_u)$. So, every connected component in G_u either belongs to $L(G_u)$ or $C(G_u)$ but is not shared between the two partitions. Thus, if a connected component contains a load or a store address node, then the connected component must be assigned to the LdSt partition because the load/store instruction is assigned to LdSt. Conversely, if a connected component contains a branch of store value and does not contain any load/store address node, then the connected component is assigned to the Comp partition.

The graph in Figure 4-4 has four connected components. One component consists of nodes $\{11^*,12,13,14^*\}$. Since this component does not contain any load/store address nodes, it can be assigned to the Comp subsystem. In contrast, all the other components contain load/store address nodes and hence are assigned to the LdSt subsystem.

The complexity of the algorithm based on reachability is $O(|V| + |E|)$ where $|V|$ is the number of nodes in the SDG and $|E|$ is the number of edges in the SDG. This directly follows from the result that the connected components of an undirected graph can be computed in $O(|V| + |E|)$ time [CLL92].

4.5 Advanced Partitioning Schemes

This section discusses advanced partitioning techniques that relax the restrictions on inserting extra instructions in order to find more computation to off-load to the Comp subsystem. The restrictions are relaxed in two ways. First, the advanced schemes assume the availability of *copy* instructions that can copy values between the LdSt and Comp register files without accessing memory. Such instructions are present in a number of ISAs (e.g. MIPS [KH92] and Alpha [Dig96]). Second, the advanced scheme duplicates some instructions to arrive at better partitions. Copy and duplicate instructions can not only increase the size of the Comp partition, but can also increase the total number of dynamic instructions executed and instruction cache miss rates. Hence, care must be taken to minimize the overheads associated with copy and duplicate instructions. Our heuristics take into account these overheads. It is shown in Section 4.6 that our heuristics introduce very few extra instructions.

```
I1:          move      $16, $0
I1':         cp_comp   $16,$f2
I2:   $L5:   lw        $f4, regs_inv_by_call
I3:          sra,c     $f4, $f4, $f2
I4:          andi,c    $f4, $f4, 0x1
I5:          beq,c     $f4, $0, $L4
I6:          move      $4, $16
I7:          jal       delete_equiv_reg
I8:          lw        $3, reg_tick
I9:          sll       $2, $16, 2
I10:         addu      $2, $2, $3
I11:         lw        $f0, 0($2)
I12:         bltz,c    $f0, $L4
I13:         addu,c    $f0, $f0, 1
I14:         sw        $f0, 0($2)
I15:  $L4:   addu      $16, $16, 1
I15':        cp_comp   $16, $f2
I16:         slt,c     $f4, $f2, 66
I17:         bne,c     $f2,$0,$L5
```

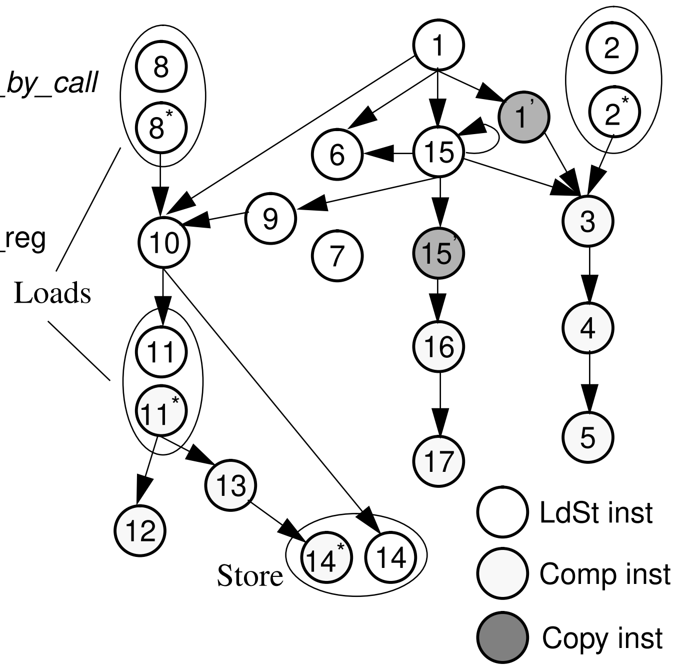


Figure 4-5. Partitioning with copies.

4.5.1 Limitations of the Basic Partitioning Scheme

The need for advanced partitioning schemes is first motivated by presenting specific examples where the basic partitioning algorithm is limited in its ability to move computation to the Comp subsystem.

Function calls limit the ability of the basic partitioning algorithm in finding Comp computation in the called function and near the call site because calling conventions require all the integer-value arguments to be passed in integer registers and the return value to be returned in an integer register. Since the basic scheme is constrained not to introduce extra (copy) instructions, all instructions at the call site that compute argument values, and all instructions inside the function that use argument values are assigned to the LdSt subsystem. The same holds for instructions that compute function return values and instructions that use function return values. One solution to this problem is to use *copy* instructions. Once could let the algorithm partition code ignoring the restrictions imposed by the calling conventions and later, when necessary, introduce copies to adhere to the conventions.

If any branch or store-value computation in the program is supplied a value by any addressing instruction, then the basic partitioning scheme assigns that computation to the LdSt subsystem. Figure 4-4 shows the SDG and the partitioning generated by the basic partitioning scheme for our running example. In the example, the branch computations {I16, I17} and {I2, I3, I4, I5} are supplied by the addressing instructions I1 and I15 and hence could not be assigned to Comp. By inserting copies for the results of I1 and I15, these branch computations can execute in Comp. Figure 4-5 shows the code generated and the associated SDG when this is done. In this example, copies have enabled the off-loading of five more instructions to Comp. Since I1' is outside the loop, copy overheads are repeatedly incurred only for node I15'.

For this example code-duplication can be used to achieve the same partitioning as realized by inserting copies. In the C code fragment shown in Figure 4-1, the loop induction variable `regno` is used both for address computation as well as for branch computation. By duplicating the induction variable `regno` in Comp, the two pieces of code can proceed independently without any communication. Figure 4-6 shows the assembly code and the associated SDG when this is done. I1' and I15' are duplicated instructions and enable five more instructions to be off-loaded to the Comp subsystem. Again, since I1' is outside the loop, duplication overheads are repeatedly incurred only for node I15'.

Thus, copy instructions and code duplication can achieve better code partitioning. However, arbitrary use of these techniques can hurt performance because copies and duplicates may introduce overhead. The advanced partitioning algorithm used by the compiler employs a cost model to identify profitable sites for copy insertion and code duplication. The cost model and the algorithm are briefly described here. Subramanya Sastry was a major contributor in designing the cost model and the advanced partitioning algorithm. They are discussed in detail by Sastry et al. [SPS98].

```
I1:          move      $16, $0
I1':         move,c    $f2, $0
I2:  $L5:    lw        $f4, regs_inv_by_call
I3:          sra,c     $f4, $f4, $f2
I4:          andi,c    $f4, $f4, 0x1
I5:          beq,c     $f4, $0, $L4
I6:          move      $4, $16
I7:          jal       delete_equiv_reg
I8:          lw        $3, reg_tick
I9:          sll       $2, $16, 2
I10:         addu      $2, $2, $3
I11:         lw        $f0, 0($2)
I12:         bltz,c    $f0, $L4
I13:         addu,c    $f0, $f0, 1
I14:         sw        $f0, 0($2)
I15: $L4:    addu      $16, $16, 1
I15':        addu,c    $f2, $f2, 1
I16:         slt,c     $f4, $f2, 66
I17:         bne,c     $f2,$0,$L5
```

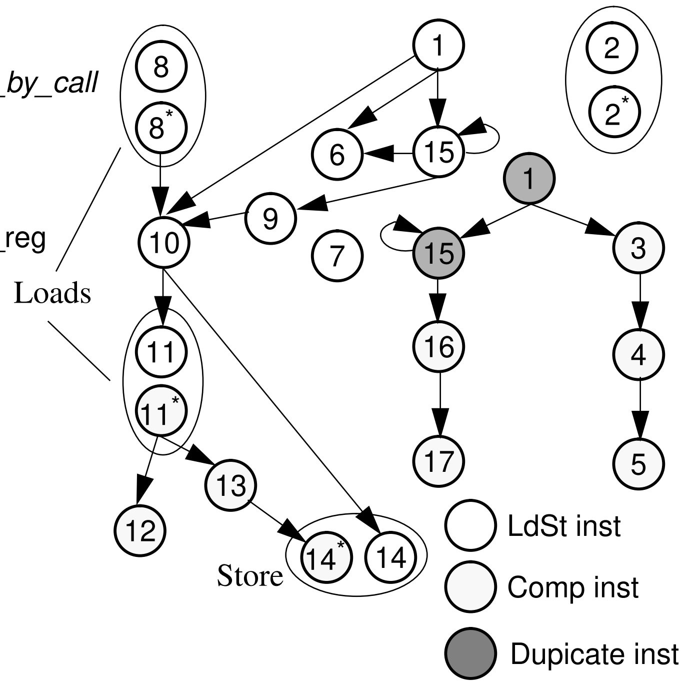


Figure 4-6. Partitioning with code duplication.

4.5.2 Cost Model

Intuitively, the benefit from a copy instruction or a duplicated instruction is the number of extra *dynamic* instructions that will execute in the Comp subsystem as a result of the copy/duplicate inserted. Symbolically, given a SDG *G*,

Let S_{copy} be the set of nodes in G for which copies are inserted.

Let S_{dupl} be the set of nodes in *G* which are duplicated

Let S_c be the set of nodes in *G* that can be moved to from LdSt to Comp as a result of the copies and duplicates.

The nodes in S_c execute in Comp yielding a bigger Comp partition. However, execution of nodes in S_{copy} and S_{dupl} introduces overhead in the program. It is beneficial to introduce these copies and duplicates only if the increase in size of the Comp partition offsets the overhead. This is quantified by the following equations.

$$Benefit = \sum_{v \in Sc} n_{B(v)}$$

$$Overhead = o_{copy} \times \sum_{v \in Scopy} n_{B(v)} + o_{dupl} \times \sum_{v \in Sdupl} n_{B(v)}$$

$$Profit = Benefit - Overhead$$

where:

B(I): Basic block containing instruction *I*

n_B: Number of times basic block *B* executed at run-time

o_{copy}: Overhead of a copy instruction

o_{dupl}: Overhead of a duplicate instructions

Hence, it is beneficial to introduce copies and duplicate instructions only if $Profit \geq 0$.

4.5.3 Algorithm for Introducing Copies and Duplicating Code

A simple heuristic is used to decide whether a given node *v* should be copied[1]or duplicated. The heuristic uses the number of parents of the node as input. The heuristic favors duplication of the node if it has few parents or if the node has parents outside its enclosing loop. In our example program, nodes 1 and 15 are candidates for copying/duplication. Because node 15 is within a loop, both techniques introduce an overhead of one instruction per loop iteration. Duplication of node 15 requires that node 1 be duplicated/copied. Because node 1 is outside the loop, duplication is preferable.

The advanced partitioning algorithm starts by initializing the LdSt partition to be the LdSt slice. Then the algorithm iteratively expands the LdSt partition to include instructions that are not profitable for execution in the Comp subsystem. It does so by analyzing the instructions on the boundary between the LdSt and Comp partitions for execution in

1. to be more precise, the result of node *v* should be copied.

the Comp subsystem. The boundary is made up of LdSt nodes whose children are not in LdSt. For each child of a boundary instruction, the algorithm essentially checks if the benefit of executing the child instruction in the Comp subsystem is positive, taking into account the extra copies and duplicate instructions that might be necessary. If not, the boundary is expanded to include the instruction in the LdSt partition. The algorithm stops when the boundary can no longer be grown. The advanced partitioning algorithm is described in further detail by Sastry et al. [SPS98].

4.6 Experimental Evaluation

4.6.1 Evaluation Methodology

We used *gcc-2.7.1* as the base compiler for studying the partitioning schemes. The compiler was modified by Subramanya Sastry to generate code for the extended SimpleScalar [BAB96] ISA which is based on the MIPS ISA. The SimpleScalar instruction set was extended by using new opcodes to encode integer instructions executing in the augmented floating-point subsystem. For the conventional microarchitecture, the benchmark programs are compiled by the base compiler (unmodified *gcc-2.7.1*).

Code partitioning is performed on the intermediate representation of the program. This is done only after the initial machine-independent optimizations [ASU88] like loop-invariant code motion, constant propagation, common subexpression elimination, etc., are complete. Register allocation is performed only after code partitioning is performed. Operands of instructions in Comp are allocated floating-point registers.

A timing simulator based on the SimpleScalar tool set [BAB96] was used for performance evaluations. The timing simulator models both a conventional and an integer-decoupled microarchitecture. Both microarchitectures are identical except for execution of integer operations in the floating-point subsystem. The simulator is cycle-based and the machine parameters simulated for the 4-way and 8-way issue machines are detailed in Table 4.2.

Parameter	4-way	8-way
Fetch width	any 4 instructions	any 8 instructions
I-Cache	32 KB, 2-way set associative 64 byte lines, 1 cycle hit time 6 cycle miss penalty	
Branch predictor	McFarling's gshare[McF93] with 1M 2-bit counters, 20bit global history, unconditional control flow instructions predicted perfectly	
Rename width	any 4 instructions	any 8 instructions
Issue window size	16 int/16 fp	32 int/ 32 fp
Max. in-flight insts	32	64
Retire width	4	8
Functional units	2 Int + 2 Fp units	4 Int + 4 Fp units
Functional unit latency	6 cycle mul, 12 cycle div, 1 cycle for rest	
Issue mechanism	up to 4 ops/cycle	up to 8 ops/cycle
	out-of-order issue loads may execute when prior store addresses are known	
Physical registers	48 int/48 fp	80 int/80 fp
D-Cache	32 KB, 2-way set-associative, write-back, write-allocate, 32 byte lines, 1 cycle hit time, 6 cycle miss penalty	
	one load/store port	two load/store ports

Table 4.2: Machine parameters.

We used programs from the SPECint95 benchmark suite to conduct our evaluation. The benchmarks and the inputs used are given in Table 4.3. The base optimization level used for compiling the benchmarks is -O3 which enables common subexpression elimination, loop invariant removal, and jump optimizations among others. All the benchmarks were run to completion. *Compress* had the lowest instruction count at 410 millions instructions and *perl* had the highest at 1.2 billion instructions.

Benchmark	Input
compress	test.in
li	browse.lsp
gcc	stmt.i
m88ksim	ctl.raw, dhry.big
go	2stone9.in
ijpeg	vigo.ppm
perl	srabbl.ppl

Table 4.3: Benchmark programs.

4.6.2 Performance Results

In this subsection, results for the performance of the two partitioning schemes and the net speedups possible with the integer-decoupled microarchitecture are presented. All our results are based on the assumption that only the simple integer operations shown in Table 4.1 are supported in the Comp subsystem. We then examine the impact on performance of supporting some of the more complex integer operations in the Comp subsystem.

Percentage of Computation Off-loaded to the Comp subsystem

The graph in Figure 4-7 shows the percentage of total dynamic instructions off-loaded by the compiler for each of the benchmark programs. The graph shows the size of the Comp partition for both the basic and the advanced partitioning schemes. Because all the benchmark programs are integer programs that execute negligible floating-point instructions, the bars in the graph correspond to the amount of integer computation that the compiler is able to identify and off-load to the Comp subsystem. Overall, the compiler is successful in off-loading a sizable fraction of the total computation to the Comp subsystem. In the case of *ijpeg*, *m88ksim*, and *gcc* more than 20% of the total computation is supported in the Comp subsystem.The graph also shows that the advanced partitioning

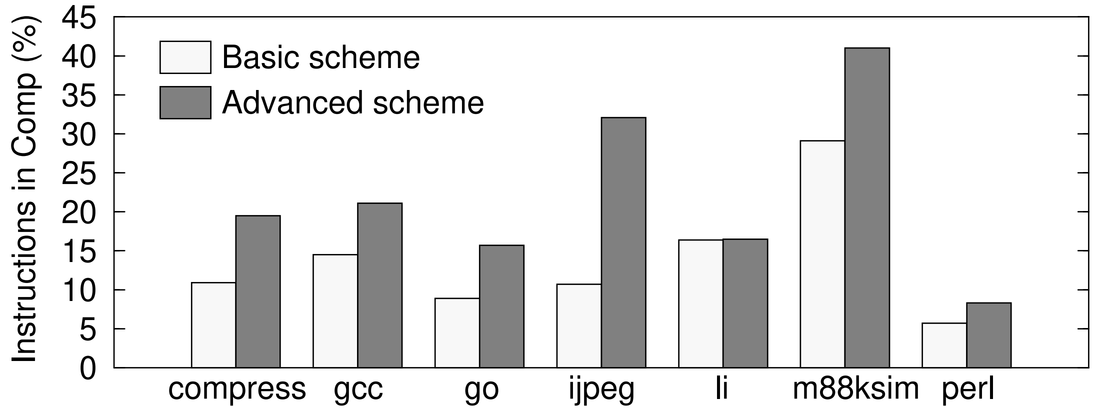


Figure 4-7. Percentage of instructions assigned to Comp.

scheme generates bigger partitions than the basic scheme for all the benchmarks. For *perl*, *go*, and *compress*, the partitions generated by the advanced partitioning scheme are almost twice the size of those generated by the basic scheme. *Ijpeg* benefits the most from the advanced scheme: the Comp computation increases from 10.7% to 32.1%. However, for *li*, the advanced scheme does not perform better than the basic scheme because *li* is call intensive and has a number of small functions.

While the advanced partitioning scheme might be able to off-load more computation, the percentages must be judged in conjunction with the change in the instruction cache performance and the total number of instructions executed due to the extra instructions introduced. Hence, we studied the overhead introduced by the advanced partitioning scheme. For all the benchmarks, we found the change in static code size to be negligible. As a result there was very little change in I-cache hit rates for all the benchmarks. Only in the case of *perl* was there a noticeable increase in I-cache hit rate by 1.8%. The increase in the number of dynamic instructions executed is also small. The maximum increase is 2% for *compress*. Copies account for 0.6% and the rest, 1.4% is due to duplicates. For *gcc*, there is a 1.2% increase in instruction count, half of which resulted from an increase in loads and stores. Copies and duplicates accounted for the rest. Overall, these results show that the advanced partitioning scheme is successful in increasing the Comp partition sizes without introducing a lot of overhead.

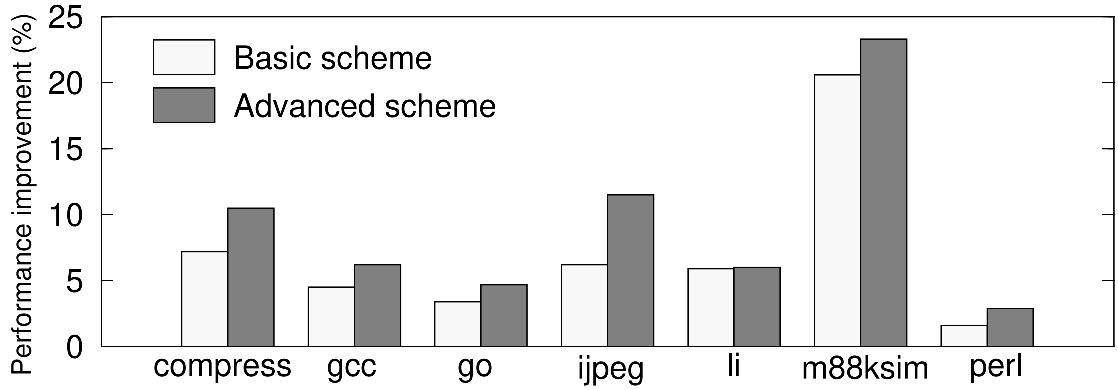


Figure 4-8. Speedups on the 4-way machine.

Performance Improvements

The graph in Figure 4-8 shows the performance improvements obtained by the integer-decoupled microarchitecture over a conventional microarchitecture for the 4-way issue (2 int + 2 fp) machine. Improvements due to both the basic and the advanced partitioning schemes are presented. For *m88ksim*, *compress*, and *ijpeg*, performance improvements over 10% are achieved with the advanced partitioning scheme. In the case of *m88ksim*, an impressive improvement of 23% is achieved with the advanced partitioning scheme. Overall for the 4-way machine, the integer-decoupled microarchitecture coupled with the advanced partitioning scheme is capable of providing modest to impressive speedups over the conventional microarchitecture.

As expected, performance improvements increase as more instructions are off-loaded to the Comp subsystem. However, the improvements do not directly reflect the size of the Comp partitions, i.e. a bigger Comp partition does not necessarily result in a greater performance improvement, for two reasons. First, the load imbalance between the LdSt and the Comp partitions results in lower speedups than expected. For example, the Comp partition of *ijpeg* with advanced partitioning is bigger than that of *m88ksim* with basic partitioning, but the corresponding improvement of *ijpeg* is much smaller than that of *m88ksim*. We found load imbalance to be the culprit in this case. There are phases in which majority of the computation is supported in the Comp subsystem leaving the LdSt subsystem rela-

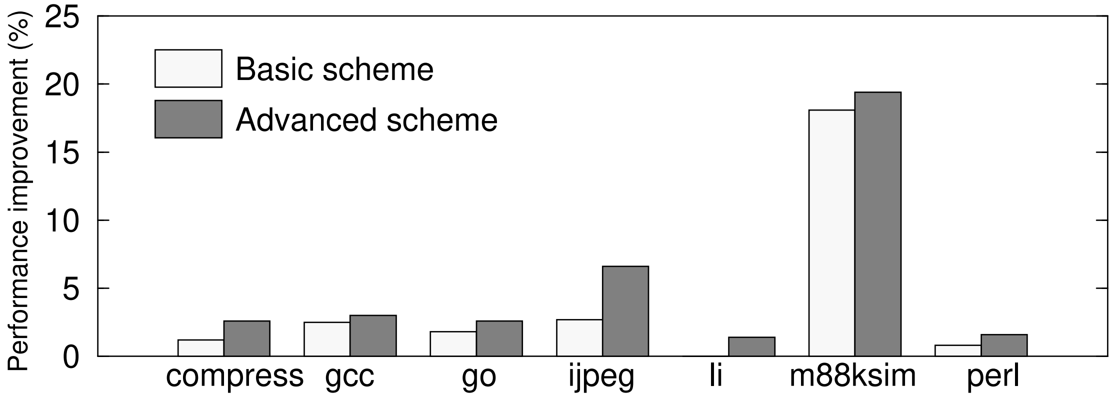


Figure 4-9. Speedups on the 8-way machine.

tively idle. Quantitatively, simulations of *ijpeg* show that the LdSt subsystem is idle 13.5% of the cycles when the Comp subsystem is executing one or more instructions. The equivalent number for *m88ksim* is only 4.4%. With the advanced partitioning scheme, *m88ksim* also suffers from the problem of load imbalance. For *m88ksim* with the advanced scheme, the LdSt subsystem is idle 12.4% of the cycles in which the Comp subsystem is executing one or more instructions. This partly explains why performance only improves by about 2.6% even though the size of the partition increases by 12%.

Another reason performance might not improve with Comp partition size is that in some cases the critical path of execution is not affected by partitioning. For example, with the basic partitioning scheme, 15% of the code in *mpegplay* executes in the Comp subsystem, but the resulting speedup is only 2.7%. Loads and stores contribute close to 47% of the total instructions in the benchmarks, and hence performance is largely determined by the cache bandwidth available. Since the integer-decoupled microarchitecture has the same cache bandwidth as the conventional microarchitecture, the performance of *mpegplay* does not improve significantly. Even with the advanced partitioning scheme and a bigger Comp partition, the speedup is only 4%.

The graph also shows that for most benchmarks, the advanced partitioning scheme yields better speedups than the basic partitioning scheme. The two exceptions are *li* and *m88ksim*. In the case of *li*, the increase in the size of the Comp partition is very small. For *m88ksim*, load imbalance seems to be the problem as mentioned earlier.

Performance Improvements on the 8-way machine

The graph in Figure 4-9 shows performance improvements on the 8-way issue (4 int + 4 fp) machine. The speedups on the 8-way issue machine are smaller than the speedups achieved on the 4-way issue machine. This is expected because the number of units in the LdSt subsystem now gets within the range of average parallelism in the programs. So, the extra issue bandwidth available in the Comp subsystem is not exploited as much. However, *m88ksim* achieves an improvement of 19% because it has enough parallelism and is able to exploit the presence of a bigger instruction window and the wider issue and execution bandwidth.

Instruction mix of the Comp partition

The instruction mix of the Comp partition, assuming that integer multiply and divide operations are also available in the Comp subsystem, is shown in Figure 4-10. The graphs shows that, except for *ijpeg*, all the benchmarks execute a negligible number of integer multiply and divide operations in the Comp subsystem. *Ijpeg* has the maximum percentage of multiplies at 2.77%. *Ijpeg* also has the maximum number of divides at 0.11%. For the remaining benchmarks, the instruction mix is almost entirely composed of simple control, logical, and arithmetic instructions. This observation matches with the results of other studies [HP96].

For *ijpeg*, we studied the performance effects of supporting integer multiply and divide operations in the Comp subsystem. This has a dramatic effect on the basic partitioning scheme. The Comp percentage increased from 11% to 40%. The speedups also increased from 6% to 16% because in some frequently executed functions of *ijpeg*, the multiply instructions are closely related to the rest of the instructions in the function. So, when the multiply instructions are moved to the LdSt subsystem, all reachable instructions are also

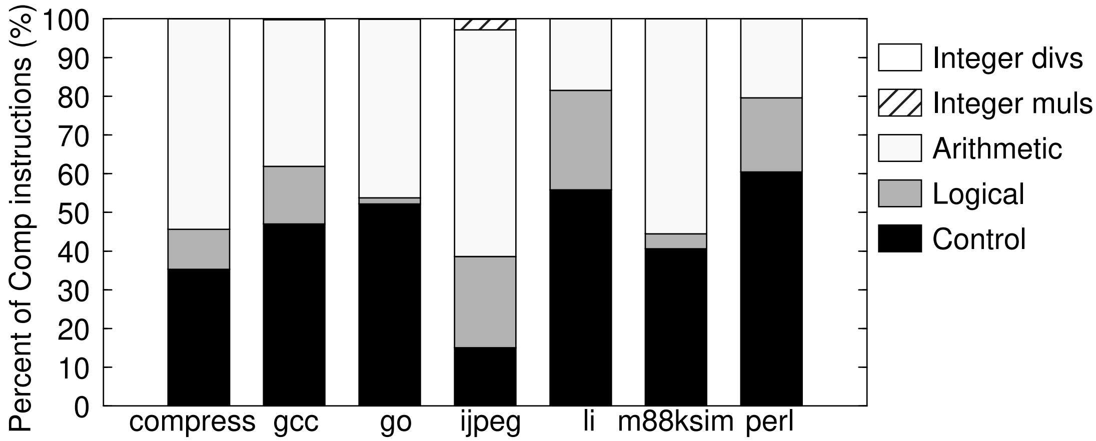


Figure 4-10. Instruction mix of the Comp partition.

moved to the LdSt subsystem which effectively moves the whole function to LdSt. However, the change was not as marked with the advanced partitioning scheme because it was able to recoup some of the computation that got moved to LdSt using copies. The Comp partition size increased from 11% to 32%. The performance improvement on the 4-way issue machine increased from 6% to 11%. This shows that the advanced partitioning scheme is successful in reducing the impact of the absence of integer multiply and divide instructions in the Comp subsystem.

4.7 Related Work

The early Control Data Corporation and Cray Research style of architectures [Rus78, Tho61] were the first to distinguish operand access and computation. One set of functional units and registers is used for addressing and a second set is used for computation in these architectures. Smith [Smi82] proposed the decoupled style of machine organization in which operand access and computation are separated and executed in parallel. The access subsystem executes memory access related instructions while the execute subsystem supports compute instructions. The access and execute subsystems communicate through queues. This organization style permits the access subsystem to slip ahead of the execute subsystem and hence, helps hide the latency of memory access. Experimental evaluation showed considerable speedups for the floating-point programs studied. Work along similar lines is reported by Pleszkun and Davidson [PD83], Goodman et al. [GHL+85], and Bird et al.[BRT93].

The decoupling concept has since been successfully implemented in a number of commercial machines like the IBM RS/6000 [Gro90] and the MIPS R8000 [Hsu94]. However, both these implementations only decouple integer and floating-point subsystems. While this helps to decouple memory access and computation in floating-point programs, integer programs cannot benefit from decoupling in these implementations.

The work presented in this chapter extends earlier work in the area of decoupled architectures in two important ways. First, the proposed integer-decoupled microarchitecture applies the concept of decoupling to integer programs. Second, decoupling is used as a technique to extract additional performance for integer codes from conventional microarchitectures without increasing their complexity.

In the context of the compiler work presented, the most closely related work is reported by Capitanio et al. [CDN92]. They study code partitioning for a VLIW architecture with partitioned register files. Their architecture consists of a number of homogeneous clusters each of which are statically scheduled. In contrast, the integer-decoupled microarchitecture is heterogeneous; only the LdSt subsystem can execute loads and stores. Further, the earlier study applied code partitioning only to straight-line loop bodies and did not consider code duplication as a means of avoiding inter-partition communication.

4.8 Chapter Summary

Conventional microarchitectures suffer from idle floating-point resources when executing integer codes. This chapter proposed integer-decoupled microarchitectures that address this drawback by supporting some of the non-addressing computation in integer programs in an augmented floating-point subsystem. For integer programs, this provides extra issue and execution bandwidth as well as provides a larger window for dynamic scheduling without increasing the complexity of the conventional microarchitecture. Furthermore, the only change required to the hardware is the implementation of simple integer operations in the floating-point subsystem.

The performance of the proposed microarchitecture was evaluated relative to a conventional microarchitecture. The results show two things. First, for the benchmarks studied, the compiler is able to off-load a significant fraction, from 9% to 41%, of the total computation in integer programs to the augmented floating-point subsystem. Second, as a result the performance improvements in the 3% to 23% range were achieved on a 4-way issue processor.

Hence, I believe that the integer-decoupled microarchitecture is an attractive choice for future processors especially considering that the hardware changes required to adapt the conventional microarchitecture are small.

Chapter 5

Conclusions

This thesis examined the trade-off between hardware complexity and clock speed in the design of superscalar microarchitectures. Using the results of the trade-off analysis, the thesis proposed and evaluated two new superscalar microarchitectures designed with the goal of achieving high performance by reducing complexity.

5.1 Thesis Summary

Superscalar microarchitectures provide high performance by using hardware techniques to execute multiple instructions every cycle. The performance of these microarchitectures is directly proportional to the product — $\textit{Instructions Per Cycle} \times \textit{Clock Frequency}$. Instructions Per Cycle or IPC measures the amount of parallelism extracted by the microarchitecture and Clock Frequency is the speed at which the microarchitecture can be clocked. Complex hardware helps improve the IPC factor by extracting higher levels of instruction-level parallelism. However, the complex hardware employed to achieve high IPC can potentially slow the clock and hence, nullify the improvements in IPC. Therefore, there is a need for developing microarchitectures that judiciously use hardware complexity for extracting higher levels of parallelism while permitting a fast clock; that is, to develop microarchitectures we refer to as *complexity-effective microarchitectures*.

To design microarchitectures that are complexity-effective, computer architects need simple models for measuring complexity that can be used at a fairly early stage of the design process. In addition to determining complexity-effectiveness, such models help identify long-term complexity trends.

The first part of this thesis presented simple models that quantifying the complexity of superscalar microarchitectures. A baseline superscalar pipeline is presented and structures whose complexity grows with increasing ILP are identified. Of these structures, register renaming, instruction window wakeup, instruction window selection, register file access, and operand bypassing are analyzed in detail. Each is modeled and Spice simulated for three different feature sizes representing past, present, and future technologies. Simple analytical models are developed that express the delay of each of the structures in terms of microarchitectural parameters like issue width and instruction window size. The impact of technology trends is also studied. In particular, the impact of poor scaling of wire delays in future technologies is analyzed.

Results show that the logic associated with managing the issue window of a superscalar processor is likely to become the most critical structure as we move towards wider-issue, larger windows, and advanced technologies in which wire delays dominate. One of the functions implemented by the logic is the broadcast of results tags over wires that span the instruction window. This operation does not scale well especially as feature sizes are reduced. Furthermore, in order to be able to execute dependent instructions in consecutive cycles — a desirable feature from the point of view of performance — the delay of the window logic should be less than a cycle.

In addition to window logic, a second structure that needs careful consideration in future technologies is the data bypass logic. The length of result wires used to broadcast bypass values increases linearly with issue width and hence, the delay of the data bypass logic increases at least linearly with issue width. As a result, the data bypass delay can grow significantly for wider microarchitectures in future technologies and force architects to consider clustered microarchitectures.

To address the complexity of window logic and data bypass logic, a family of complexity-effective microarchitectures called the dependence-based superscalar microarchitectures is proposed and studied. The proposed microarchitectures achieve the dual goals of high IPC and a fast clock using two main techniques. The machine is partitioned into multiple clusters each of which contains a slice of the instruction window and execution resources of the whole processor. This enables high-speed clocking of the individual clusters since the narrow issue width and the small instruction window in each cluster keeps critical delays small. The second technique involves intelligent steering of instructions to the multiple clusters so that the whole width of the machine is utilized while minimizing the performance degradation due to slow inter-cluster communication. Experimental results show that dependence-based superscalar microarchitectures are capable of extracting similar levels of parallelism as conventional microarchitectures while facilitating a faster clock.

The third contribution of this thesis is the integer-decoupled microarchitecture. The integer-decoupled microarchitecture improves the performance of integer programs and can be integrated into a conventional microarchitecture with little or no increase in complexity. Floating-point units in the conventional microarchitecture are augmented to perform simple integer operations and the resulting floating-point subsystem is used to support some of the computation in integer programs. The computation to be off-loaded is identified by the compiler. Simulation results are presented that show modest speedups for a 4-way processor. The speedups diminish with increasing issue width.

5.2 Future directions

5.2.1 Quantifying the Complexity of Superscalar Microarchitectures

Analysis similar to that presented in this thesis can be applied to other structures in the pipeline that are not studied here. Two specific examples are the instruction fetch logic and the load/store queue logic. The complexity of the latter in particular has been problematic [Yea97] for designers in industry.

5.2.2 Dependence-based Superscalar Microarchitectures

The instruction steering heuristics studied in this thesis are simple in that they do not require more than one extra pipe stage. One avenue for future research is the feasibility and applicability of caching steering information. Caching steering information can help move the steering logic out of the critical path. This would open up the possibility of more complex steering heuristics. Therefore, it might be worthwhile to study sophisticated steering heuristics that can further boost the parallelism extracted by the dependence-based microarchitectures.

The fifo steering heuristic studied in this thesis steers instructions solely based on register dependences between instructions. It might be possible to augment the heuristic with the memory-dependence prediction techniques proposed by Moshovos et al. [MBVS97] to help create longer chains. For example, a load instruction can be steered to the fifo that contains an earlier store instruction to the same address as the one referenced by the load. Note that at the time of steering, the addresses referenced by the load and the store instruction are not known. Memory-dependence prediction can be used to chain dependent load-store pairs and steer them to the same fifo.

5.2.3 Integer-decoupled Microarchitecture

There is always scope for more research in developing improved partitioning heuristics that can off-load more computation to the augmented FP subsystem. Another possibility is to study heuristics that not only try to off-load sizable fraction of the total computation, but also try to balance the load on the two subsystems.

An alternative scheme for utilizing the idle floating-point subsystem in a conventional microarchitecture, is to use the idle subsystem to execute along both paths of likely mispredicted branches [HS96] in integer programs. Of course, this would require extra hardware support.

References

[ABHS89] M. C. August, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger. Cray X-MP: The Birth of a Supercomputer. *IEEE Computer*, 22:45–54, January 1989.

[ACR95] P. S. Ahuja, D. W. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[AMG+95] C. Asato, R. Montoye, J. Gmuender, E. W. Simmons, A. Ike, and J. Zasio. A 14-Port 3.8ns 116-Word 64b Read-Renaming Register File. In *1995 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 104–105, February 1995.

[AS92] T. M. Austin and G. S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, May 1992.

[Ass97] Semiconductor Industry Association. The National Technology Roadmap for Semiconductors, 1997.

[AST67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11:8–24, January 1967.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison Wesley, 1988.

[BAB96] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: the SimpleScalar Tool Set. Technical Report CS-TR-96-1308 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, July 1996.

[Boh95] M. T. Bohr. Interconnect Scaling - The Real Limiter to High Performance ULSI. In *1995 International Electron Devices Meeting Technical Digest*, pages 241–244, 1995.

[BP92] M. Butler and Y. N. Patt. An Investigation of the Performance of Various Dynamic Scheduling Techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 1–9, December 1992.

[Bre] S. E. Breach. Design and Evaluation of a Multiscalar Processor. Ph.D. thesis in preparation at University of Wisconsin–Madison.

[BRT93] P. L. Bird, A. Rawsthorne, and N. P. Topham. The Effectiveness of Decoupling. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 47–56, 1993.

[Buc62] W. Bucholtz. *Planning a Computer System: Project Stretch*. McGraw-Hill, 1962.

[CDN92] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register FIles for VLIWs: A Preliminary Analysis of Tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, December 1992.

[Cha81] A. E. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. *IEEE Computer*, 14(9):18–27, 1981.

[Cha91] T. Chappell. A 2ns Cycle, 4 ns Access 512kb CMOS ECL SRAM. In *1991 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 50–51, February 1991.

[Cha95] J. I. Chamdani. *Microarchitecture Techniques to Improve the Design of Superscalar Microprocessors*. PhD thesis, Georgia Institute of Technology, March 1995.

[CLL92] T. H. Cormen, C. E. Leiserson, and R. L.Rivest. *Introduction to Algorithms*. McGraw Hill, 1992.

[CNO+88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Transactions on Computers*, 37:9667–979, August 1988.

[DF90] P. K. Dubey and M. J. Flynn. Optimal Pipelining. *Journal of Parallel and Distributed Computing*, 8:10–19, 1990.

[Dig96] Digital Equipment Corporation. *Alpha Architecture Handbook, Version 3*, October 1996.

[DM74] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Dataflow Computer. In *Proceedings of the 2nd Annual International Symposium on Computer Architecture*, pages 126–132, 1974.

[D+74] R. Dennard et al. Design of Ion-implanted MOSFETs With Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, SC-9:256–268, 1974.

[D+92] D. Dobberpuhl et al. A 200-MHz 64-bit Dual-issue Microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11), November 1992.

[DT92] H. Dwyer and H. C. Torng. An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, December 1992.

[Ell85] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, Massachussets, 1985.

[FCJV97] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, December 1997.

[Fis81] J. A. Fisher. Trace Scheduling: A Technique For Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7), July 1981.

[FJC96] K. I. Farkas, N. P. Jouppi, and P. Chow. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[FJD80] S. Fuller, A. Jones, and I. Durham. CMU Cm* Review. Technical Report AD-A050135, Department of Computer Science , Carnegie-Mellon University, 1980.

[Fra93] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin–Madison, November 1993.

[FS92] M. Franklin and G. S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–69, May 1992.

[GHL+85] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Plezkun, P. B. Schechter, and H. C. Young. PIPE: A VLSI Decoupled Architecture. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 20–27, 1985.

[Gro90] G. F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[G+97] B. A. Gieseke et al. A 600MHz Superscalar RISC Microprocessor With Out-Of-Order Execution. In *1997 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 176–177, February 1997.

[Gwe93] L. Gwennap. Speed Kills? Not for RISC Processors. *Microprocessor Report*, 7(3):3, March 1993.

[Gwe95a] L. Gwennap. Hal Reveals Multichip SPARC Processor. *Microprocessor Report*, 9(3), March 1995.

[Gwe95b] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2), February 1995.

[Gwe95c] L. Gwennap. UltraSparc Adds Multimedia Instructions. *Microprocessor Report*, 8(16), December 1995.

[Gwe96a] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14):11–16, October 1996.

[Gwe96b] L. Gwennap. Intel's MMX Speeds Multimedia. *Microprocessor Report*, 10(3), March 1996.

[HF88] I. S. Hwang and A. L. Fisher. A 3.1ns 32b CMOS Adder in Multiple Output Domino Logic. In *1988 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 140–141, February 1988.

[Hin95] G. Hinton. Pentium Pro Processor, December 1995. Tutorial talk at 28th Annual International Symposium on Microarchitecture.

[HP86] W. W. Hwu and Y. N. Patt. HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 297–307, June 1986.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[HPS92] M. Horowitz, S. Przybylski, and M. D. Smith. Recent Trends in Processor Design: Reclimbing the Complexity Curve, August 1992. Tutorial talk at Western Institute of Computer Science, Stanford University.

[HS96] T. H. Heil and J. E. Smith. Selective Dual Path Execution. Unpublished, University of Wisconsin-Madison, November 1996.

[Hsu94] P. Y. T. Hsu. Design of the R8000 Microprocessor. *IEEE Micro*, pages 23–33, April 1994.

[I$^+$95] Inoue et al. A 0.4um 1.4ns 32b Dynamic Adder Using Non-precharge Multiplexers and Reduced Precharge Voltage Techniques. In *1995 Symposium on VLSI Circuits Digest of Technical Papers*, pages 9–10, June 1995.

[JJ90] M. G. Johnson and N. P. Jouppi. Transistor model for a Synthetic 0.8um CMOS process, May 1990. Class notes for Stanford University EE371.

[Joh91] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

[Jol91] R. D. Jolly. A 9-ns, 1.4-Gigabyte/s, 17-Ported CMOS Register File. *IEEE Journal of Solid-State Circuits*, 26(10), October 1991.

[JW89] N. P. Jouppi and D. W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Kel96] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, October 1996. 9th Annual Microprocessor Forum, San Jose, California.

[KF96] G. A. Kemp and M. Franklin. PEWS: A Decentralized Dynamic Scheduler for ILP Processing. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 239–246, 1996.

[KH92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KM89] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit Microprocessor. In *IEEE Micro*, pages 15–30, August 1989.

[Kog81] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.

[KS86] S. R. Kunkel and J. E. Smith. Optimal Pipelining in Supercomputers. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986.

[K$^+$93] P. Knebel et al. HP's PA7100LC: A Low-Cost Superscalar PA-RISC Processor. In *Proceedings of COMPCON*, pages 441–448, 1993.

[Kum96] A. Kumar. The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor. In *Proceedings of the Hot Chips VIII*, pages 9–20, August 1996.

[LW92] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[Mat97] D. Matzke. Will Physical Scalability Sabotage Performance Gains. *IEEE Computer*, 30(9):37–39, 1997.

[MBVS97] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[McF93] S. McFarling. Combining Branch Predictors. Technical Report DEC WRL Technical Note TN-36, DEC Western Research Laboratory, 1993.

[Met87] Meta-Software Inc. *HSpice User's Manual*, June 1987.

[MF95] G. McFarland and M. Flynn. Limits of Scaling MOSFETs. Technical Report CSL-TR-95-662 (Revised), Stanford University, November 1995.

[NH97] K. Nowka and H. P. Hofstee. Circuits and Microarchitecture for Gigahertz VLSI Designs. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 284–287, September 1997.

[Now95] K. Nowka. *High-Performance CMOS System Design Using Wave Pipelining*. PhD thesis, Stanford University, August 1995.

[PD83] A. R. Pleszkun and E. S. Davidson. Structured Memory Access Architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 461–471, 1983.

[PS81] D. A. Patterson and C. H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[Rab96] J. M. Rabaey. *Digital Integrated Circuits - A Design Perspective*. Prentice Hall Electronics and VLSI Series, 1996.

[RBS96] E. Rotenberg, S. Bennet, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proccedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[RF72] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.

[RJSS97] E. Rotenberg, Q. Jacobson, Q. Sazeides, and J. E. Smith. Trace Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, December 1997.

[RNOM95] K. Rahmat, O. S. Nakagawa, S-Y. Oh, and J. Moll. A Scaling Scheme for Interconnect in Deep-Submicron Processes. Technical Report HPL-95-77, Hewlett-Packard Laboratories, July 1995.

[Rus78] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[RYYT89] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *IEEE Computer*, 22:12–35, January 1989.

[SBV95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[Sch71] H. Schorr. Design Principles for a High-Performance System. In *Symposium on Computers and Automata, Polytechnic Institute of Brooklyn*, pages 165–192, 1971.

[SDC95] S. P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC Microprocessor. In *IEEE Micro*, pages 8–17, October 1995.

[SF91] G. S. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[Smi82] J. E. Smith. Decoupled Access/Execute Computer Architecture. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 112–119, April 1982.

[Smi95] J. E. Smith. New Paradigms for Instruction Level Parallelism, October 1995. Talk prepared at University of Wisconsin–Madison.

[Smi97] J. E. Smith. Amdahl's Law: Not Just an Equation, June 1997. Keynote speech at the 24th Annual International Symposium on Computer Architecture.

[Soh90] G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Units, Pipelined Computers. *IEEE Transactions on Computers*, 39:349–359, March 1990.

[SP88] J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37:562–573, May 1988.

[SPS98] S. Sastry, S. Palacharla, and J. E. Smith. Exploiting Idle Floating-Point Resources for Integer Execution. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[SS90] J. E. Smith and G. S. Sohi. Studies in Program Characteristics and Architectural Choices for High-Performance, Fine-Grain Parallel Processors, 1990. Grant proposal prepared at University of Wisconsin–Madison.

[SS95] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, December 1995.

[S+87] J. E. Smith et al. The ZS-1 Central Processor. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, October 1987.

[S+91] H. Shinohara et al. A Flexible Multiport RAM Compiler for Data Path. *IEEE Journal of Solid-State Circuits*, 26(3), March 1991.

[S+93] M. Suzuki et al. A 1.4ns 32b CMOS ALU in Double Pass-Transistor Logic. *IEEE Journal of Solid-State Circuits*, 28(11), November 1993.

[TF70] G. S. Tjaden and M. J. Flynn. Detection and Parallel Execution of Independent Instructions. *IEEE Transactions on Computers*, C-19:889–895, October 1970.

[Tho61] J. E. Thornton. Parallel Operation in the Control Data 6600. In *Proceedings of the Fall Joint Computers Conference*, volume 26, pages 33–40, 1961.

[Tho63] J. E. Thornton. Considerations in Computer Design – Leading up to the CONTROL DATA 6600, 1963. Contol Data Chippewa Laboratory Report.

[Tom67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[T+96] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[Unk79] Unknown. CRAY-2 Central Processor, 1979. Unpublished Cray Research Report.

[VM97] S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, June 1997.

[V+96] N. Vasseghi et al. 200 MHz Superscalar RISC Processor Circuit Design Issues. In *1996 IEEE International Sold-State Circuits Conference Digest of Technical Papers*, pages 356–357, February 1996.

[WE93] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, second edition, 1993.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[Wil51] M. V. Wilkes. The Best Way to Design an Automatic Calculating Machine. In *Proceedings of the Manchester University Computer Inaugural Conference*, pages 16–18, July 1951.

[Wil95] N. C. Wilhelm. Why Wire Delays Will No Longer Scale for VLSI Chips. Technical Report SMLI TR-95-44, Sun Microsystems Laboratories, August 1995.

[WJ94] S. J. E. Wilton and N. P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, DEC Western Research Laboratory, July 1994.

[WO95] K. M. Wilson and K. Olukotun. High Performance Cache Architectures to Support Dynamic Superscalar Microprocessors. Technical Report CSL-TR-95-682, Stanford University, June 1995.

[WRP92] T. Wada, S. Rajan, and S. A. Przybylski. An Analytical Access Time Model for On-Chip Cache Memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.

[Yea96] K. C. Yeager. Mips R10000 Superscalar Microprocessor. In *IEEE Micro*, April 1996.

[Yea97] K. C. Yeager, October 1997. Personal Communication.

[YP92] T. Y. Yeh and Y. N. Patt. Alternate Implementations of Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.

Appendix A

A.1 Technology Parameters

The Hspice Level 3 models used to simulate the synthetic 0.8µm, 0.35µm, and 0.18µm CMOS technologies are given in Table A.1.

Parameter	0.8µm	0.35µm	0.18µm
tox	165	70	35
vto	0.77(-0.87)	0.67(-0.77)	0.55(-0.55)
uo	570(145)	535(122)	450(80)
gamma	0.8(0.73)	0.53(0.42)	0.40(0.32)
vmax	2.7e5(0.0)	1.8e5(0.0)	1.05e5(0.0)
theta	0.404(0.233)	0.404(0.233)	0.404(0.233)
eta	0.04(0.028)	0.024(0.018)	0.008(0.008)
kappa	1.2(0.04)	1.2(0.04)	1.2(0.04)
phi	0.90	0.90	0.90
nsub	8.8e16(9.0e16)	1.38e17(1.38e17)	4.07e17(4.07e17)
nfs	4e11	4e11	4e11
xj	0.2µ	0.2µ	0.2µ
cj	2e-4(5e-4)	5.4e-4(9.3e-4)	10.6e-4(21.3e-4)
mj	0.389(0.420)	0.389(0.420)	0.389(0.420)
cjsw	4e-10	1.5e-10	3.0e-11
mjsw	0.26(0.31)	0.26(0.31)	0.26(0.31)
pb	0.80	0.80	0.80
cgso	2.1e-10(2.7e-10)	1.8e-10(2.4e-10)	1.8e-10(2.4e-10)
cgdo	2.1e-10(2.7e-10)	1.8e-10(2.4e-10)	1.8e-10(2.4e-10)
delta	0.0	0.0	0.0
ld	0.0001µ	0.0001µ	0.0001µ
rsh	0.5	0.5	0.5
Vdd	5.0	2.5	2.0

Table A.1: Spice parameters.

Table A.2 gives the metal resistance and capacitance values assumed for the three technologies.

Technology	R_{metal} ($\Omega/\mu m$)	C_{metal} ($fF/\mu m$)
0.8µm	0.02	0.275
0.35µm	0.046	0.628
0.18µm	0.09	1.22

Table A.2: Metal resistance and capacitance.

A.2 Delay Results

Issue Width	Decoder Delay (ps)	Wordline Drive Delay(ps)	Bitline Delay(ps)	Total Delay(ps)
0.8µm technology				
2	540.3	218.9	498.2	1502.2
4	547.1	227.9	529.6	1566.9
8	562.5	245.8	594.2	1700.9
0.35µm technology				
2	220.2	95.6	236.5	649.4
4	225.8	103.9	259.2	698.5
8	243.1	115.8	303.1	800.8
0.18µm technology				
2	129.6	70.6	175.7	435.4
4	136.8	78.2	193.4	478.9
8	148.4	92.5	228.5	561.7

Table A.3: Break down of rename delay.

Window Size	Tag Drive Delay(ps)	Tag Match Delay(ps)	Match OR Delay(ps)	Total Delay(ps)
Issue Width = 2				
8	73.0	331.3	248.1	652.4
16	82.6	333.1	248.5	664.2
24	92.6	337.3	248.8	678.7
32	103.7	344.0	249.1	696.9
40	114.9	347.7	248.9	711.5
48	126.3	352.4	248.7	727.5
56	137.4	358.7	249.2	745.4
64	149.1	364.6	248.7	762.4
Issue Width = 4				
8	74.5	368.2	407.0	849.7
16	86.4	372.4	406.8	865.6
24	98.8	377.6	403.9	880.3
32	112.3	384.8	409.2	906.2
40	126.2	392.3	408.7	927.2
48	140.6	400.1	404.2	944.9
56	156.3	409.0	404.1	969.4
64	172.4	416.9	403.3	992.7
Issue Width = 8				
8	77.5	400.2	665.3	1143.0
16	93.3	406.6	665.7	1165.5
24	111.4	415.2	664.8	1191.4
32	130.7	425.2	658.5	1214.4
40	151.5	437.7	660.2	1249.5
48	174.4	451.0	658.3	1283.8
56	199.3	465.0	664.6	1328.9
64	228.2	479.2	664.6	1372.0

Table A.4: Break down of window wakeup delay for 0.8μm technology.

Window Size	Tag Drive Delay(ps)	Tag Match Delay(ps)	Match OR Delay(ps)	Total Delay(ps)
Issue Width = 2				
8	28.5	126.1	101.3	255.8
16	33.4	128.7	101.5	263.7
24	38.3	129.1	101.2	268.6
32	43.7	133.2	97.3	274.1
40	49.7	136.3	101.2	287.3
48	53.1	138.8	97.4	289.3
56	58.9	142.7	101.1	302.8
64	64.4	145.0	98.9	308.3
Issue Width = 4				
8	29.7	147.1	155.8	332.6
16	36.0	151.2	158.3	345.4
24	42.7	155.0	159.1	356.8
32	50.5	157.7	158.4	366.7
40	56.3	163.2	159.0	378.5
48	63.2	168.1	159.6	390.9
56	72.0	171.9	157.0	400.9
64	80.9	179.0	159.1	419.0
Issue Width = 8				
8	32.2	173.4	257.6	463.2
16	41.6	177.5	257.8	476.9
24	51.1	183.7	257.8	492.5
32	61.9	190.6	257.7	510.1
40	74.7	199.1	257.7	531.5
48	88.8	208.9	257.6	555.3
56	102.9	216.4	258.4	577.7
64	121.8	224.8	258.4	605.0

Table A.5: Break down of window wakeup delay for 0.35µm technology.

Window Size	Tag Drive Delay(ps)	Tag Match Delay(ps)	Match OR Delay(ps)	Total Delay(ps)
Issue Width = 2				
8	14.6	67.9	60.7	143.1
16	18.8	68.7	60.6	148.1
24	22.4	69.8	60.6	152.7
32	26.1	71.8	60.6	152.7
40	29.9	73.6	60.3	163.8
48	33.7	75.7	59.9	169.3
56	36.6	77.3	61.0	174.8
64	41.4	79.4	59.7	180.5
Issue Width = 4				
8	15.8	84.1	84.7	184.7
16	21.1	85.1	84.4	190.6
24	26.1	87.6	84.8	198.5
32	31.2	90.8	84.3	206.3
40	36.6	93.3	84.8	214.7
48	41.7	96.5	84.4	222.5
56	47.5	99.4	84.8	231.8
64	54.1	102.8	84.4	241.3
Issue Width = 8				
8	18.8	104.9	123.6	247.3
16	26.1	108.4	123.8	258.3
24	33.8	113.6	123.1	270.5
32	42.0	118.2	125.0	285.1
40	51.5	124.8	123.2	299.5
48	62.6	130.4	123.0	316.0
56	75.1	135.2	123.2	333.4
64	90.0	139.4	122.9	352.3

Table A.6: Break down of window wakeup delay for 0.18µm technology.

Window Size	$T_{reqpropd}$(ps)	T_{root}(ps)	$T_{grantpropd}$(ps)	Total Delay(ps)
0.8µm technology				
16	233.2	607.2	272.5	1113.0
32	532.5	737.6	727.4	1997.5
64	534.6	742.9	719.8	1997.4
128	802.8	753.4	1118.5	2674.6
0.35µm technology				
16	125.0	338.5	135.4	598.9
32	246.6	339.7	295.4	881.7
64	245.5	338.0	296.3	879.8
128	347.9	338.5	460.3	1146.7
0.18µm technology				
16	53.6	141.7	55.1	250.4
32	107.0	141.2	123.5	371.7
64	106.9	144.2	121.9	373.0
128	159.9	146.7	195.5	502.1

Table A.7: Break down of selection delay.

Issue Width	Window Size	Register File Size	Rename Delay(ps)	Window Delay(ps)	Register File Delay(ps)	Data Bypass Delay(ps)
2	16	48	1374.57	1777.20	1902.05	233.15
4	32	80	1417.25	2903.70	2222.10	411.12
8	64	120	1489.91	3369.4	2715.71	836.79

Table A.8: Overall delay results for 0.8μm technology.

Issue Width	Window Size	Register File Size	Rename Delay(ps)	Window Delay(ps)	Register File Delay(ps)	Data Bypass Delay(ps)
2	16	48	524.76	862.60	724.43	110.45
4	32	80	554.08	1248.40	873.21	223.79
8	64	120	603.59	1484.80	1155.45	486.50

Table A.9: Overall delay results for 0.35μm technology.

Issue Width	Window Size	Register File Size	Rename Delay(ps)	Window Delay(ps)	Register File Delay(ps)	Data Bypass Delay(ps)
2	16	48	285.43	398.50	393.43	91.00
4	32	80	311.55	578.00	498.29	177.58
8	64	120	355.62	725.30	729.40	421.42

Table A.10: Overall delay results for 0.18μm technology.

Appendix B

The constants in the delay equations presented in Chapter 2 are tabulated below. The table entries contain both absolute and relative values of the constants. The relative values are presented to show how each component's contribution varies with feature size.

B.1 Register Rename Logic

Decoder delay

$$T_{decoder} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8μm	387.16 (1.00)	8.611 (2.22e-2)	1.07e-2 (2.76e-5)
0.35μm	153.66 (1.00)	5.425 (3.53e-2)	1.07e-2 (6.96e-5)
0.18μm	81.88 (1.00)	3.96 (4.84e-2)	1.07e-2 (1.31e-5)

Table B.1: Constants in decoder delay equation for rename logic.

Wordline delay

$$T_{wordline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8µm	98.71 (1.00)	7.17 (7.26e-2)	1.93e-3 (1.96e-5)
0.35µm	39.18 (1.00)	4.52 (1.15e-1)	1.93e-3 (4.92e-5)
0.18µm	20.88 (1.00)	3.30 (1.58e-1)	1.93e-3 (9.24e-5)

Table B.2: Constants in wordline delay equation for rename logic.

Bitline delay

$$T_{bitline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8µm	525.75 (1.00)	22.06 (4.19e-2)	5.84e-3 (1.11e-2)
0.35µm	208.67 (1.00)	13.90 (6.67e-2)	5.84e-3 (2.80e-2)
0.18µm	111.20 (1.00)	10.14 (9.12e-2)	5.84e-3 (5.25e-2)

Table B.3: Constants in bitline delay equation for rename logic.

Total delay

$$T_{rename} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8µm	1011.62 (1.00)	37.84 (3.74e-2)	1.78e-2 (1.76e-5)
0.35µm	401.51 (1.00)	23.84 (5.94e-2)	1.78e-2 (4.43e-5)
0.18µm	213.96 (1.00)	17.40 (8.13e-2)	1.78e-2 (8.32e-5)

Table B.4: Constants in total delay equation for rename logic.

B.2 Window Wakeup Logic

Tag drive delay

$$T_{tagdrive} = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c_5 \times IW^2) \times WINSIZE^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)	c_3 (ps)	c_4 (ps)	c_5 (ps)
0.8µm	18.14 (1.00)	6.37e-1 (3.51e-2)	9.43e-2 (5.20e-3)	3.05e-3 (1.68e-4)	1.52e-3 (8.38e-5)	1.21e-4 (6.67e-6)
0.35µm	7.20 (1.00)	2.97e-1 (4.12e-2)	5.94e-2 (8.25e-3)	2.10e-3 (2.92e-4)	1.29e-3 (1.79e-4)	1.21e-4 (1.68e-5)
0.18µm	3.84 (1.00)	1.82e-1 (4.74e-2)	4.34e-2 (1.13e-2)	1.66e-3 (4.32e-4)	1.08e-3 (2.81e-4)	1.21e-4 (3.15e-5)

Table B.5: Constants in tag drive delay equation for wakeup logic.

Tag match delay

$$T_{tagmatch} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8µm	390.68 (1.00)	6.01 (1.54e-2)	3.35e-3 (8.57e-6)
0.35µm	83.15 (1.00)	3.48 (4.18e-2)	3.35e-3 (4.03e-5)
0.18µm	45.46 (1.00)	2.55 (5.61e-2)	3.35e-3 (7.37e-5)

Table B.6: Constants in tag match delay equation for wakeup logic.

Match OR delay

$$T_{matchOR} = c_0 + c_1 \times IW$$

Feature Size	c_0 (ps)	c_1 (ps)
0.8µm	60.00	70.00
0.35µm	26.25	30.62
0.18µm	13.63	15.75

Table B.7: Constants in match OR delay equation for wakeup logic.

Total delay

$$\begin{aligned} T_{wakeup} &= (c_0 + c_1 \times IW + c_2 \times IW^2) \\ &+ (c_3 + c_4 \times IW) \times WINSIZE \\ &+ (c_5 + c_6 \times IW + c_7 \times IW^2) \times WINSIZE^2 \end{aligned}$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)	c_3 (ps)	c_4 (ps)	c_5 (ps)
0.8µm	468.82 (1.00)	76.01 (1.62e-1)	3.35e-3 (7.14e-6)	6.37e-3 (1.36e-5)	9.43e-2 (2.01e-4)	3.05e-3 (6.50e-6)
0.35µm	116.60 (1.00)	34.10 (2.91e-1)	3.35e-3 (2.87e-5)	2.97e-1 (2.55e-3)	5.94e-2 (5.09e-4)	2.10e-3 (1.80e-5)
0.18µm	62.93 (1.00)	18.30 (2.91e-1)	3.35e-3 (5.32e-5)	1.82e-1 (2.89e-3)	4.34e-2 (6.90e-4)	1.66e-3 (2.64e-5)

Feature Size	c_6 (ps)	c_7 (ps)
0.8µm	1.52e-3 (3.24e-6)	1.21e-4 (2.58e-7)
0.35µm	1.29e-3 (1.11e-5)	1.21e-4 (1.04e-6)
0.18µm	1.07e-3 (1.70e-5)	1.21e-4 (1.92e-6)

Table B.8: Constants in total delay equation for wakeup logic.

B.3 Window Selection Logic

$$T_{select} = c_0 + c_1 \times \log_4 WINSIZE$$

Feature Size	c_0 (ps)	c_1 (ps)
0.8µm	127.61	322.51
0.35µm	50.65	128.00
0.18µm	26.99	68.21

Table B.9: Constants in total delay equation for selection logic.

B.4 Register File Logic

Decoder delay

$$T_{decoder} = c_0 + (c_1 + c_2 \times IW) \times NPREG + (c_3 + c_4 \times IW + c_5 \times IW^2) \times NPREG^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)	c_3 (ps)	c_4 (ps)	c_5 (ps)
0.8µm	414.62 (1.00)	9.63e-2 (2.32e-4)	2.03e-2 (4.89e-5)	1.94e-6 (4.68e-9)	4.37e-6 (1.05e-8)	2.46e-6 (5.93e-9)
0.35µm	164.56 (1.00)	6.06e-2 (3.68e-4)	2.03e-2 (1.23e-4)	1.94e-6 (1.18e-8)	4.37e-6 (2.65e-8)	2.46e-6 (1.49e-8)
0.18µm	87.69 (1.00)	4.43e-2 (5.05e-4)	2.03e-2 (2.31e-4)	1.94e-6 (2.21e-8)	4.37e-6 (4.98e-8)	2.46e-6 (2.80e-8)

Table B.10: Constants in decoder delay equation for register file logic.

Wordline delay

$$T_{wordline} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8µm	203.92 (1.00)	49.66 (2.43e-1)	1.61e-1 (7.90e-4)
0.35µm	80.94 (1.00)	31.29 (3.86e-1)	1.61e-1 (1.99e-3)
0.18µm	43.13 (1.00)	22.84 (5.30e-1)	1.61e-1 (3.73e-3)

Table B.11: Constants in wordline delay equation for register file logic.

Bitline delay

$$T_{bitline} = c_0 + (c_1 + c_2 \times IW) \times NPREG + (c_3 + c_4 \times IW + c_5 \times IW^2) \times NPREG^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)	c_3 (ps)	c_4 (ps)	c_5 (ps)
0.8µm	300.00 (1.00)	1.02 (3.40e-3)	2.54e-1 (8.47e-4)	1.02e-5 (3.40e-8)	1.40e-5 (4.67e-8)	2.85e-6 (9.50e-9)
0.35µm	119.07 (1.00)	4.05e-1 (3.40e-3)	1.60e-1 (1.34e-3)	6.40e-6 (5.37e-8)	8.80e-6 (7.39e-8)	2.85e-6 (2.39e-8)
0.18µm	63.45 (1.00)	2.96e-1 (4.66e-3)	1.17e-1 (1.84e-3)	4.68e-6 (7.38e-6)	6.42e-6 (1.01e-7)	2.85e-6 (4.49e-6)

Table B.12: Constants in bitline delay equation for register file logic.

Total delay

$$\begin{aligned} T_{regfile} &= (c_0 + c_1 \times IW + c_2 \times IW^2) \\ &+ (c_3 + c_4 \times IW) \times NPREG \\ &+ (c_5 + c_6 \times IW + c_7 \times IW^2) \times NPREG^2 \end{aligned}$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)	c_3 (ps)	c_4 (ps)	c_5 (ps)
0.8µm	918.53 (1.00)	49.66 (5.41e-2)	1.61e-1 (1.75e-4)	1.12 (1.22e-3)	2.74e-1 (2.98e-4)	1.21e-5 (1.32e-8)
0.35µm	364.57 (1.00)	31.29 (8.58e-2)	1.61e-1 (4.42e-4)	4.65e-1 (1.28e-3)	1.80e-1 (4.94e-4)	8.34e-6 (2.29e-8)
0.18µm	194.27 (1.00)	22.84 (1.18e-2)	1.61e-1 (8.29e-4)	3.40e-1 (1.75e-3)	1.37e-1 (7.05e-4)	6.62e-6 (3.41e-8)

Feature Size	c_6 (ps)	c_7 (ps)
0.8µm	1.84e-5 (2.00e-8)	5.31e-6 (5.78e-9)
0.35µm	1.32e-5 (3.62e-8)	5.31e-6 (1.46e-8)
0.18µm	1.08e-5 (5.56e-8)	5.31e-6 (2.73e-8)

Table B.13: Constants in total delay equation for register file logic.

B.5 Data Bypass Logic

$$T_{bypass} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Feature Size	c_0 (ps)	c_1 (ps)	c_2 (ps)
0.8µm	18.13 (1.00)	25.50 (1.41)	6.15 (0.34)
0.35µm	7.20 (1.00)	16.06 (2.23)	6.15 (0.85)
0.18µm	3.84 (1.00)	11.72 (3.06)	6.15 (1.60)

Table B.14: Constants in total delay equation for data bypass logic.