Coprocessor Architectures for VLSI

By

Paul Mark Hansen

B.S. (Utah State University) 1972
M.S. (Utah State University) 1974

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: D. A. Patterson 11/18/88
Chair Carlo H. Séquin Date 11/11/88
R. W. Brodersen 11/21/88

****

Coprocessor Architectures for VLSI

by

Paul Mark Hansen

COPROCESSOR ARCHITECTURES FOR VLSI

Ph.D.
Computer Science

Paul Mark Hansen

Electrical Engineering
and Computer Sciences

Abstract

The hardware resources available on a single chip to implement VLSI CPU remain scarce, despite rapid technological advances. Reduced Instruction Set Computers (RISCs) reduce complexity and use the chip hardware resources to make the most frequently occurring operations fast. In this dissertation, the RISC philosophy is extended to specialized devices called *coprocessors*. Coprocessors increase system performance by reducing the number of instructions per program and the number of effective cycles per instruction.

A method for evaluating coprocessor performance is developed, including a model that accounts for system, software, and hardware effects. Coprocessor implementations are characterized in terms of *effectiveness* and *utilization* by considering operation and overhead time for typical computations.

Performance and interface characteristics of the SPUR floating-point coprocessor implementation of the IEEE Standard are presented and compared to two popular commercial versions by Intel and Motorola. The SPUR FPU is a factor of three to 50 times faster than the commercial versions for comparable technology and clock rates. For each architecture, the influence on performance of each of the following is identified: the bus width between the floating-point unit and operand storage, the operand transfer protocols implemented in hardware, the concurrent execution model, the speed of the function units, the floating-point instruction semantics, and the data cache service time. Execution time spent in overhead is shown to increase to more than 90% for some architectures if equipped with faster floating-

point units. This suggests that coprocessor interface architectures must change dramatically to keep pace with the rapid advance in CPU execution rates to be effective.

The combinatorial optimization problem of finding the shortest path between two vertices in a directed graph is presented. Algorithms for scan-based relaxation techniques and Dijkstra's shortest-path algorithm are considered in detail. A path optimization coprocessor based on the SPUR model is proposed that achieves nearly three orders of magnitude improvement in performance over software implementations, and two to three orders of magnitude improvement in cost with performance comparable to dedicated hardware devices or specialized and multicomputer architectures.

Finally, the SPUR coprocessor architecture is evaluated for three other applications: digital signal processing, vector floating-point arithmetic, and support for the Prolog language.

D.A. Patterson

David A. Patterson
Chairman of Committee

Dedicated with love to Kathleen, Angela, Michael, Brian, and Carolyn

Acknowledgments

I am grateful to many individuals for valuable contributions that made this dissertation possible. First and foremost are my wife, Kathleen, and children, Angela, Michael, Brian, and Carolyn. Without their cheerful support and love through the years at Berkeley, this work simply could not have been completed. As well, I am grateful to my parents, the late Vernon L. Hansen and Loretta J. Hansen, and other family members for their love and concern.

I thank Dave Patterson, my advisor and committee chairman, for support, encouragement, and guidance on many matters. Dave has amazing capacity and strength of purpose, and succeeds in making work fun — characteristics worth emulating.

I thank the other members of my committee, Carlo Séquin and Ronald Wolff, for the timely help in reading and approving this dissertation. I am particularly indebted to Carlo for providing invaluable suggestions and insights concerning Chapter 5.

Others at Berkeley have had an interest in and influence on what is reported here, and I would like to thank them as well: Velvel Kahan for several thought-provoking discussions, some of which were about floating-point arithmetic; Paul Hilfinger for sharing some ideas on path optimization algorithms; Shing Kong for details of the coprocessor interface with the SPUR CPU; and BK Bose and George Taylor for insights about floating-point, the implementation of the SPUR FPU, and some late-night philosophizing.

I relied on the help and friendship of many individuals in industry to support my studies. In particular, I thank Ken Shoemaker at Intel and Jim Peek and Ralph Campbell at Cimlinc for letting me consume their equipment to gather some of the data for Chapter 4; John Hansen and Doug Helmuth at FMC for many helpful discussions about autonomous vehicles; and Mark Ellson, Jim Caratozollo, and Frank Cirimele at Lockheed for machine time to complete the first

SPUR cache analyses, as well as other work over a period of several years.

I am grateful for many friends that have made our stay in Berkeley memorable. Among them are Ricki Blau, Scott Baden, and Erling Wold — long-time and in some cases past residents of 508-2 Evans. Also, Mike and Carol Carey deserve a special thanks. Lastly, many others, both family and friends too numerous to mention, should be thanked for never tiring of asking when I would finish.

I gratefully acknowledge the financial support provided by a Tektronix Foundation Scholarship and a research assistantship with the SPUR project. Principal funding for SPUR was provided by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269.

Table of Contents

List of Figures

List of Tables

<This page is intentionally blank.>

1 Introduction and Problem Statement

Rapid advances in the underlying technologies have enabled the computer industry to offer tremendous improvements in system performance over the past two decades. As a result, problems that a short time ago were considered intractable or computationally too expensive to solve are being handled easily by today's systems.

With this increased capability comes increased expectation. As systems provide faster solutions for todays problems, new applications are considered that demand even faster and more capable systems. The expansion of current problems in precision and magnitude continues to demand more and more power and speed. It seems a never ending battle, much like the predicament of the sorcerer's apprentice, where the most recently defeated foe spawns two in its place [Ewer27]. Likewise, the so-called *wheel of reincarnation* [Siew82] suggests that this trend is not likely to end soon.

While capacity and raw performance have been increasing at an exponential rate, the price paid for it has decreased, but not at the same pace. Often an increase in system performance is accompanied by a proportional increase in price, maintaining a constant price/performance ratio over time. For example, Figure 1-1 illustrates the price versus performance over the past decade for a popular family of minicomputers [Digi87, Dong87]. It would be nice to improve the price/performance ratio by either reducing the effective cost per quantum of calculation, or improving the performance while maintaining the same price.

One possible solution to the problem is to continue to make faster general purpose computers by riding the "technology wave" — reimplement the same architecture with faster components. However, that is not as simple as it might seem, and is likely to be a costly and time consuming process. A modern day minicomputer or advanced microprocessor system, without any consideration for the software needed, can consume man-centuries in design and implementation time [Clar86] including "CPU-centuries" in simulation time [Latt82]. As a consequence, price often tracks the increase in speed.

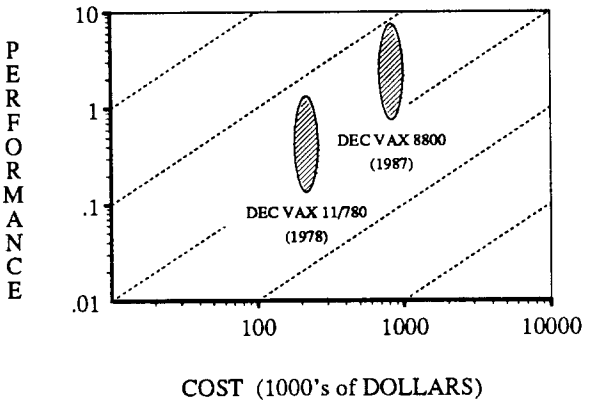


Figure 1-1. Performance versus Price.

This figure illustrates how performance per dollar has remained constant over time for the DEC family of VAX minicomputers. The diagonal lines represent constant performance per second per dollar. The shaded regions reflect the variation between the speed of floating-point arithmetic (toward the bottom of the oval) and non-floating-point computation (toward the top of the oval).

Improvements in software performance could help the situation. Much is being done to increase software productivity (i.e., the process of creating programs using CASE techniques — computer aided software engineering), but only relatively small performance improvements in execution time can be achieved with even the most sophisticated compiler and code generation techniques [Corb88]. More efficient algorithms are continually being researched and discovered, which would also help. Nevertheless, the flexibility afforded by software implementations of many functions is obtained at the price of minimum performance when compared to hardware implementations.

So, what else can be done? In the past, to *leap frog* the status quo and make substantial improvements in the performance of traditional von Neumann architectures, optional special-purpose hardware devices have been added. Such devices perform specific tasks previously done by software. We call these optional devices *coprocessors*, and include such things as attached processors, array processors, floating-point accelerators, data channels, graphics display processors, string scanner/sorters, input/output controllers, and so on. (Many of these are termed *Special Algorithm Processors* in [Siew82].) Coprocessors are becoming increasingly popular in solving the problems of expanding expectation in a cost effective and timely manner.

It is our thesis that for traditional computer architectures, optional coprocessors offer maximum performance improvement potential with minimum cost and manageable complexity and design time. Also, that coprocessors can effectively provide incremental performance improvement applicable to single instructions, important kernel functions, or entire programs. They also offer a simple programming model, often being completely transparent to the programmer or user at the application level.

This dissertation examines the role of coprocessors in traditional computer systems and discusses several aspects of coprocessor architectures and their performance implications. We show that cost effective coprocessor architectures are those that perform frequent and essential aspects of their problem domain quickly and efficiently, leaving other less frequent operations to software. This philosophy follows the reduced instruction set computer (RISC) research being pursued at the University of California at Berkeley and elsewhere. Simplifications achieved with this approach allow for rapid specification and design and reduced complexity in the implementation phase, yielding a more cost effective solution to the problems considered.

We evaluate single-chip integrated circuit coprocessor implementations designed to work with single-chip CPU microprocessors, and focus on how the interface between the coprocessor and its host influences its performance and effectiveness. Careful consideration must be given to how the coprocessor interfaces with the rest of the system (both physical interconnection and the software model). Otherwise, many inherent advantages of having a second execution element can be lost to various overhead

factors. The coprocessor is an effective means of providing tremendous computing power in conventional systems with minimum cost and difficulty.

Contemporary designs are capitalizing on the rapid advances being made in computer-aided design of complex integrated circuits. The recently announced 164,000-transistor Motorola 88100 CPU chip was designed using silicon compiler technology in 20 calendar months and the 750,000-transistor cache/memory-management unit in 11 calendar months. Experienced designers reported a 5- to 10-fold increase in productivity using such CAD tools [Goer88]. Such advances make it possible and affordable to use silicon technology to replace particular software functions.

Sometimes highly specialized and dedicated hardware devices can exceed the performance of a general purpose RISC computer coupled with a coprocessor. The added performance typically comes with the disadvantages of higher cost and longer design time. We consider such devices only as a point of comparison to our work.

1.1. The Context of Our Research — SPUR

SPUR (Symbolic Processing Using RISCs) is a multiprocessor workstation developed at Berkeley as a research vehicle for studying symbolic and scientific computation using parallel processors [Hill86]. The project involves research in IC technology, computer architecture, operating systems, and programming languages. Figure 1-2 is a simplified block diagram of a SPUR system.* Besides the CPU and cache controller (CC), the architecture supports a floating-point unit coprocessor (FPU) which implements in hardware the basic functions of the ANSI/IEEE Standard P754-1985 for binary floating-point arithmetic. The FPU serves as a test case coprocessor for the analyses reported in this dissertation. Extensions to that model supporting other functions are also reported here.

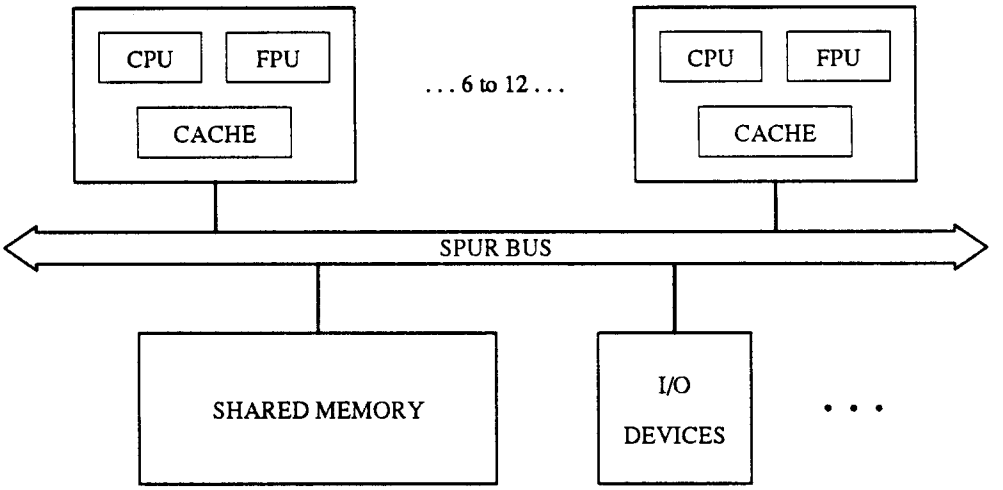


Figure 1-2. SPUR Workstation System.

This figure shows a SPUR workstation system. Up to 12 processor nodes share a common bus, memory, and I/O devices. Each node contains a central processing unit (CPU), a 128 K-byte direct-mapped mixed I&D cache controlled by a dedicated chip (CC), and a floating-point unit coprocessor (FPU). The FPU supports the ANSI/IEEE Standard P754-1985 for binary floating-point arithmetic. The SPUR BUS is a modified NuBUS supporting standard I/O and memory devices and special protocols for a cache coherency scheme implemented in hardware. The system is targeted to support research in parallel processing and Lisp environments, but the architectural features added for that support do not impede the execution of other standard languages, such as C. A network operating system, Sprite, is also under development by researchers at Berkeley [Oust88].

*A more detailed diagram of the SPUR processor node is included in Chapter 4.

1.2. Overview of Results

We have investigated the implementation of the SPUR FPU in conjunction with the SPUR CPU to determine if the combination can provide good performance for IEEE floating-point arithmetic. We have compared our approach with other contemporary implementations and have identified several factors that contribute to or detract from good performance. For fundamental floating-point computations — dot product, Gaussian elimination, and others — we report that the speed of our CPU-FPU pair exceeds the performance of software implementations by two orders of magnitude, and some commercial implementations by factors of three to 50. Researchers at Berkeley have implemented the three VLSI chips, with the CPU and cache controller running successfully in the lab. The FPU has been fabricated and is waiting its turn for further testing in the system.

We have also investigated whether a coprocessor optimized for dynamic programming functions can yield a cost effective solution to computation and memory intensive problems, such as path optimization. Since no hardware coprocessor implementations of this function are known, we have made comparisons with general purpose and special purpose machines running software implementations. Highly specialized architectures designed specifically for path optimization and CPU architectures with structure amenable to the systolic-like operations useful for path optimization were also considered. A path optimization coprocessor has been proposed and evaluated as an alternative to both software and specialized hardware. Our results show nearly three orders of magnitude improvement in performance over software implementations, and two to three orders of magnitude improvement in cost at comparable performance when compared to dedicated hardware or specialized CPU architectures.

Features of our coprocessor interface architecture that contribute to good performance include: parallel instruction decoding between the CPU and coprocessor, a direct connection between the coprocessor and cache memory, a wide data path to the coprocessor to facilitate efficient transfer of all types of operands, concurrent operand loads and stores during coprocessor execution, parallel operation between the CPU and coprocessor (i.e., the CPU can continue executing instructions while the coprocessor is busy), and synchronization mechanisms between the CPU and coprocessor that either are implicit or program controlled. The FPU microarchitecture uses several algorithmic and circuit design techniques that lead to good performance.

1.3. Dissertation Organization

Chapter 2 considers some of the advances in technology that have caused the explosive growth in computer performance over the past two decades, and the implications for the future of coprocessors and application-specific architectures. We outline our motivations for the study, look at previous coprocessor implementations, and consider contemporary work in coprocessor research.

In Chapter 3, we develop coprocessor classifications based on their interaction with the system, from both a software and hardware perspective. Our method for performance evaluation used in subsequent chapters is explained.

Chapter 4 examines floating-point arithmetic coprocessors and describes in detail the work carried out in the evaluation of the SPUR FPU implementation at Berkeley.

Chapter 5 considers in depth the algorithmic needs of a path optimization coprocessor. Various alternatives in the fundamental algorithms are simulated extensively to determine the best algorithms, implementation techniques, and data structures for coprocessors designed to perform the most essential and time critical functions of shortest-path optimization.

Chapter 6 briefly considers three other applications — digital signal processing, vector floating-point processing, and support for the Prolog language — to determine if they are suitable for coprocessor implementations using the SPUR model.

Chapter 7 summarizes this work, identifies the contributions, and suggests future work in the area of coprocessor research.

The appendices include data from some of the analyses completed during research reported in this dissertation. Appendix A is a condensed specification of the coprocessor interface for the SPUR FPU

implementation. A more complete version is found in [Hans86]. Appendix B includes the C-language and SPUR assembly language versions of benchmarks used in Chapter 4. The simulated performance of commercial floating-point coprocessors and the SPUR FPU is tabulated. The instruction times for commercial floating-point coprocessors investigated in this dissertation are summarized. Appendix C includes path optimization simulation results showing the performance of various algorithms investigated in Chapter 5, and a brief overview of the programs used to generate the data for this dissertation. Appendix D includes information about digital signal processor technology supporting the research reported in Chapter 6.

<This page is intentionally blank.>

2 Motivation, Background, and Related Work

2.1. Introduction and Overview

This chapter considers factors that support the notion of using coprocessors in general purpose computer systems. We illustrate the evolution of ideas with several examples, specifically in VLSI systems — the focus of our research. Section 2.2 discusses the motivation to balance system performance. Section 2.3 describes how technological advances, architectural innovations, and component cost reductions will influence future coprocessor development. Section 2.4 describes some successful non-VLSI coprocessor applications used in past and current systems. Section 2.5 focuses on VLSI coprocessors, particularly floating-point and memory-management units, and broadly surveys other commercial devices. Section 2.6 summarizes the ideas covered in this chapter to set the stage for the analysis presented in Chapter 3.

2.2. Motivation

Many tasks performed on traditional von Neumann architecture computers are *resource-bound.* For example, an intensive floating-point computation may cause the system to become *processor bound.* Alternatively, the system may become *I/O bound* with a compiler that reads and writes many temporary disk files during its work. The net result is that some system resources are not being used effectively, since they stand idle while waiting for other resources.

Kuck [Kuck78] illustrates the interplay between *processor bandwidth*, *memory bandwidth*, and *I/O bandwidth* in determining the *system capacity surface*, a 3-dimensional representation of the limits to performance imposed by the system components. His discussion suggests that computer systems have been tuned to provide a balance between the capabilities of different components of the system. For example, an I/O bottleneck to the swap disk can be removed by the addition of more main memory. In other cases, performance can be improved simply by employing additional execution units to allow tasks to execute in parallel, instead of in serial. Such a system might be made even more cost effective if the additional execution units are tailored to the task, i.e., coprocessors. The identification of which

tasks should be implemented in a coprocessor is a critical step. Migrating an arbitrary selection of software into hardware can result in reduced overall performance or simply replacing software releases by costly engineering redesigns [Ditz81, Hans83]. If wisely used, however, the coprocessor can be an extremely effective means to achieve computer system balance. In the next section, we see how technology may influence the use of coprocessors in future computer systems.

2.3. Technology — History and Implications

Implementation technology is directly related to cost/performance effectiveness of computer systems. Advances in the speed and capacity of hardware components used to build computers have largely been responsible for the astounding growth in computer system performance over the past two decades. The electronic integrated circuit (IC) is the essential hardware building block of these systems. Since the 1960's when IC's were first used in digital computers, their complexity has evolved from fewer than 10 gates per chip (small scale integration — SSI), through several 10's of gates per chip (medium scale integration — MSI) to several hundred gates per chip (large scale integration — LSI). Today, 1000's of gates are integrated on a single chip, giving rise to the term very large scale integration, or VLSI [Texa76]. Since IC manufacturers are able to put more and more functions on each chip, computer manufacturers are able to offer more and more capability per system [Burg84].

The exponential growth in IC's was first observed by Gordon E. Moore in 1964 [Siew82]. He predicted that the number of components per chip would double every year, which held true from 1959 when the planar transistor was invented until the early 1970's. It is primarily this swift progress in IC technologies that has fostered the fast-paced computer evolution. During the past 17 years, the number of transistors per chip for memories have increased by more than a factor of 1000 (from approximately four thousand to four million devices per chip). This growth can be expressed as (adapted from [Fagg77] and Figure 5 in [Myer86]):

$$\text{Growth factor for IC memories} = 2^{\left\lceil \frac{\log_2(4M/4K)}{17} \right\rceil} \approx 1.5 \text{ per year}$$

which represents a doubling every 20 months. During the same period of time, microprocessors have increased in transistor count by a factor of 250:

$$\text{Growth factor for IC microprocessors} = 2^{\left\lceil \frac{\log_2(.5M/2K)}{17} \right\rceil} \approx 1.4 \text{ per year.}$$

Figure 2-1 plots this pattern of growth by noting when some specific memory and microprocessor components became available during those years.

Two fundamental reasons account for these gains: first, advances made in photolithography and etching technology have accounted for about a hundred-fold reduction in the minimum feature size that can be fabricated economically. Second, the fabrication techniques and computer aided design and manufacturing tools that have been developed allow about a 10-fold increase in the maximum die size that can be economically fabricated. This is illustrated in Figure 2-2. The invention of new structures and circuit cleverness has contributed to this advance, as well.

Despite fundamental density limits (about 0.25 micron minimum feature size [Bacc84]), we can still look forward to achieving about two orders of magnitude increase in density and slightly less than one order of magnitude increase in speed for silicon semiconductors [Asai86]. Naturally, other technologies are being explored to allow the rapid progress in digital electronics to continue [Vlah88]. In the next section, we consider how innovations in architecture have coupled with this technology to achieve better performance.

2.3.1. Architecture — The Implications

As one result of the changing technology, computer system performance, as gauged by the number of instructions executed per second, has also been increasing at an exponential rate. However, during the early 1980's, several researchers published studies showing that the increased capacity of silicon technology had provided an easy means to increase the complexity of systems. But, as more and

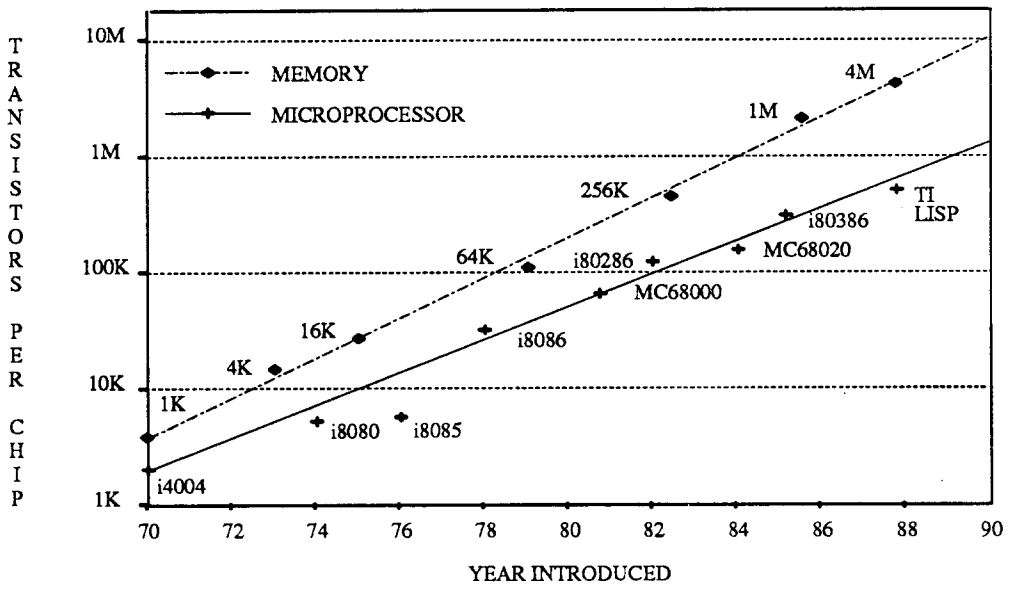


Figure 2-1. Transistors per Chip — 1970 to 1988.

This figure shows how memory and microprocessor technology have improved to provide rapid and continuous growth in the number of transistors per chip for nearly two decades. DRAM technology currently provides 4-Mbit parts in the laboratory [Mash87], with 256-Kbit static and 1-Mbit dynamic parts in common use. State-of-the-art microprocessors use more than a half-million transistors for single-chip CPU's [Boss87]. While microprocessors have increased in the number of transistors per chip by a factor of 250 during the past 17 years, memories have increased by more than a factor of 1000 [Fagg77, Myer86].

more transistors became available, they were being used less and less effectively [Ditz80, Henn82, Patt80, Radi82]. In view of the emerging VLSI technology at the time, the number of transistors, even though large and increasing, was still a limited resource, and needed to be used for the most important functions [Kate83]. Likewise, the interface to the outside world was constrained by the number of pins on VLSI chip carriers. It was very important that the internal micro-architecture reuse operands a frequently as possible to reduce off-chip interactions with memory.

To maximize the utility of these limited resources, a design style employing simplified instruction sets and pipelined single-cycle execution units emerged. The reduced instruction set computer (RISC), based on this notion of reduced complexity, resulted in fewer clock cycles per effective instruction executed. Figure 2-3 illustrates the past and projected future trends for uniprocessor system performance. For the decade since 1978, the number of *standard MIPS** (millions of instructions per second) has been doubling about every two years. For the next eight to 10 years, that growth is projected to increase at a more rapid pace due to advances in technology coupled with architecture innovations. According to Bill Joy [Joy85], the rate of growth in performance for single chip computers during the next decade is expected to follow the trend shown in Equation (2-1):

$$MIPS = 2^{Year - 1984} \qquad (2\text{-}1)$$

This far exceeds the growth rate for the decade prior to 1984, when minicomputers improved at a rate of 20% per year and microcomputers at about 40% per year [Bell84]. In the next section, we examine how the advances in technology and architecture combine to affect the costs of computing.

*A DEC VAX 11/780 has been *defined* as being a 1-MIP machine. Hence the term "standard MIP" usually refers to a performance comparison relative to VAX 11/780 execution. For example, a machine that runs a program in half the time used by a VAX 11/780 is a *2-MIP machine*.

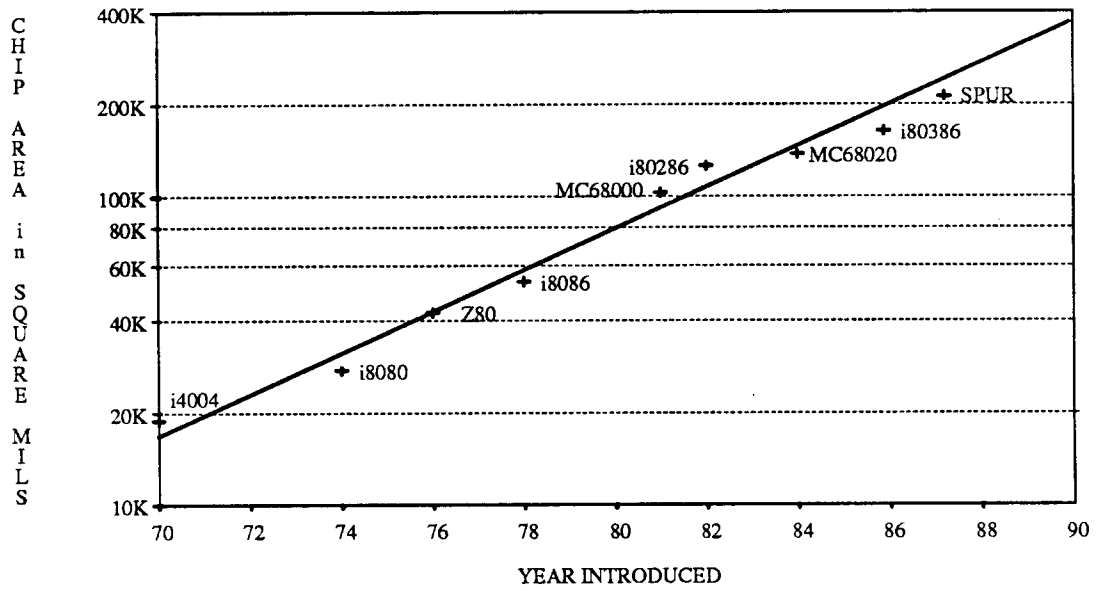


Figure 2-2. Chip Area — Microprocessors 1970 to 1988.

This figure shows how the size of integrated circuits has increased from less than 20,000 sq-mils to more than 200,000 sq-mils over the past two decades. Early microprocessors, as exemplified by the Intel i4004, i8080, and Zilog Z80 were limited to 4- or 8-bit architectures. During the mid 1970's, 16-bit architectures were developed to take advantage of the increased capacity. Finally, 32-bit architectures were introduced with the Motorola MC68000 in 1981. With an increasing number of transistors per chip, rather than scale to wider word-width architectures, additional functions such as instruction caches and memory management units have been integrated directly on the CPU chip lately. (Adapted from Figure 5 in [Myer86].)

2.3.2. Cost — The Implications

The technological advances mentioned earlier have had a considerable impact on the cost of components used to build computers. The average selling price of random-access memory storage has decreased from slightly less than $.015 per bit in 1970 to roughly $.00001 per bit in 1988 (a reduction of about 40% per year, or three orders of magnitude overall) while speed has increased by a factor of 10. The per-transistor cost of microprocessors has experienced a similar, albeit slightly less dramatic, reduction in cost of about 25% per year during the same time period [Myer86]. Nevertheless, Figure 1-1 illustrated how cost/performance has remained relatively constant over the past decade for at least one computer family. Figure 2-4 shows how a broad spectrum of machines compare, from low-end personal computers to the most expensive super-computers. The diagonal lines indicate planes of equal cost/performance [Dong87, Dong88]. It is interesting to note that most systems lie within a narrow range of price/performance (between one and eight floating-point operations per second per dollar), from the least expensive to the most expensive machines. The general systems that are the best are microprocessor-based systems equipped with floating-point coprocessors. Nevertheless, special-purpose hardwired devices, such as array processors and especially DSP chips are *orders of magnitude* better. When the amount of time consumed for a computation becomes prohibitive, the cost of high end machines can be justified. However, we are interested in the region of the design space that provides the best of both worlds — large performance improvement at lower cost. We believe the best way to achieve that is with coprocessors.

In the next section, we summarize the motivations for having coprocessors in contemporary and future computer systems

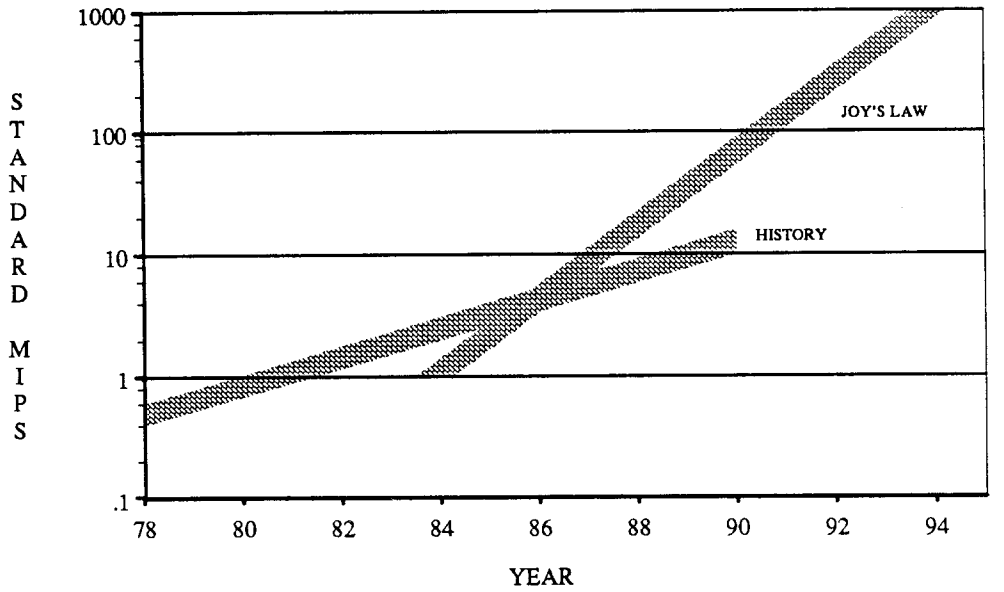


Figure 2-3. Computer System Performance in MIPS.

This figure shows how the performance of conventional minicomputer systems has progressed over the past decade. In the late 1970's and early 1980's, implementations were typically discrete MSI/LSI and multiple board CPU's. In the mid-1980's, with single-chip VLSI microprocessors becoming the computing engines, the exponential trend has continued, approximately doubling performance every two years. Architecture innovations coupled with advances in technology are projected to offer even steeper growth in CPU power over the next decade by doubling every year — twice the rate of earlier generations.

2.3.3. Putting It All Together — The VLSI Coprocessor

Performance increases brought about by technology and architecture are impressive. The reduction in effective cost per component per chip is also encouraging. With the advent of the Mead-Conway style of VLSI design [Mead80], simple abstractions of fundamental design rules and circuit design can be employed to realize successful silicon by more and more people. VLSI presents a mature medium for the realization of additional execution elements. As VLSI design gets easier, many special purpose chips will be invented, designed, and fabricated. And, as VLSI design capabilities are enhanced through more sophisticated processing techniques, smaller geometries, and so on as shown above, functions that previously were too costly to build will become affordable. Some examples of this include speech processing chips [Kava86], image processing enhancements [Clar82, Ruet86], and a host of floating-point coprocessor units [Cass84, Digi85, Inte81a, Nati84, Rowe88, Zilo83]. These and others will be discussed in more detail in later chapters.

The advantages of using VLSI to implement new systems are readily apparent: small physical size, low system power consumption, customizable design (the designer is not limited to what is available in a data book), fewer chips per system, and so forth. Nevertheless, the advantages that VLSI represents for computer designers must be viewed in perspective. It is important to be aware of the problems and pitfalls that are inherent in the process. Interestingly, though, some of the disadvantages actually work in favor of coprocessors being added to the system.

First, history has taught us that we will very likely *always* have more ideas than transistors on a single chip to accommodate those ideas. In other words, the number of transistor per chip is still a finite resource, and must be used judiciously. A coprocessor becomes a way to realize an idea without sacrificing other essential features.

Second, the design and testing of a VLSI device is still an involved and painstaking process. Contributing to that complexity is the fact that the engineering tools needed to design and produce "monster chips" are the most difficult challenge for CAD. By maintaining some physical as well as

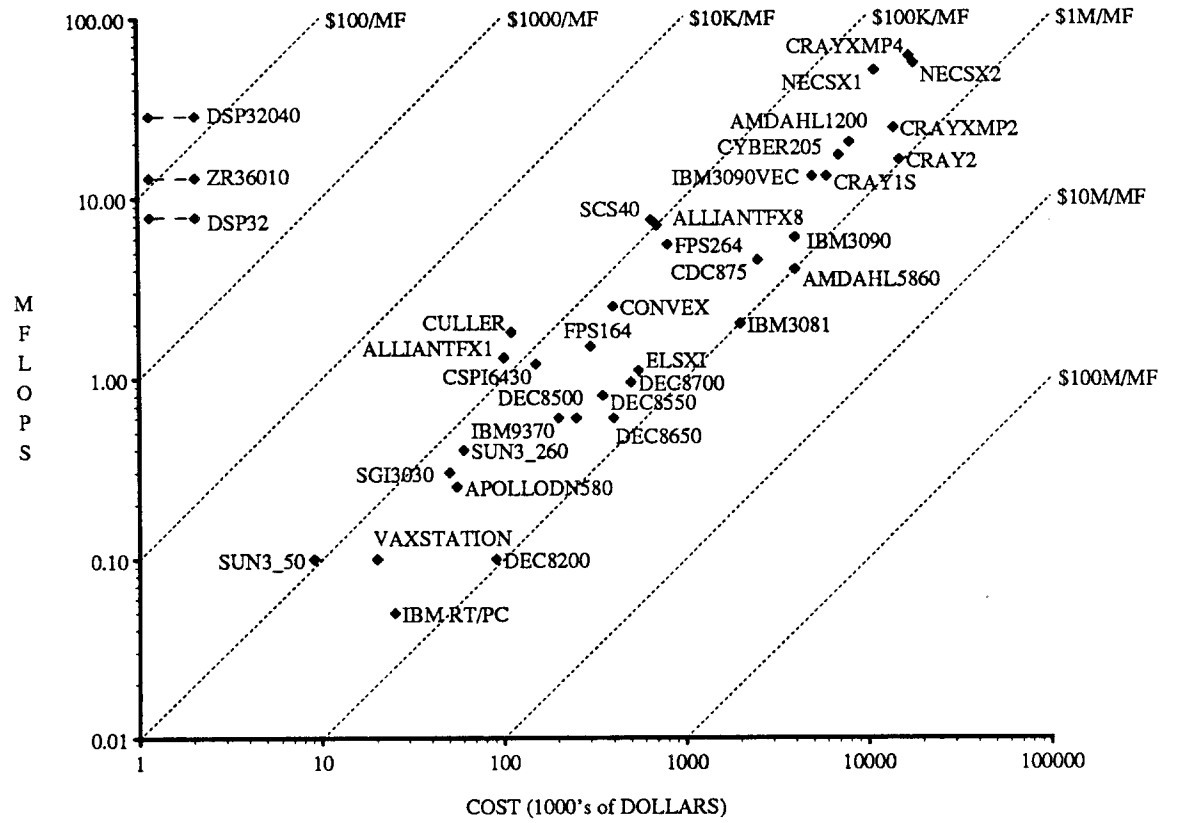


Figure 2-4. Computer System Cost Performance.

This figure shows how the cost-performance of conventional systems compares to special purpose coprocessors and specialized architectures for floating-point computation. As illustrated, high-performance general purpose systems are also high-cost. However, architectures specialized for a particular task or type of processing generally achieve the same or comparable high performance but with orders of magnitude less cost. The upper-left corner of the diagram is the region considered by our research — maximum performance but at minimum cost. (Adapted from [Dong87, Dong88].)

functional partitioning, those limitations can be avoided.

Third, at any time it is possible to find important problems that seem to require an order of magnitude more speed than provided by the state of the art [Fuss84]. But once technology has reached the limits of uniprocessor performance, significant improvements in speed can come only from exploiting parallelism in the processing tasks. Coprocessors offer a natural means of achieving concurrent execution through either pipelined or parallel architectures.

Fourth, designers will simply not anticipate many of the applications that our processors will be used for during their product lifetimes. In the past, new problems could only be accommodated by adding software routines. However, if our systems are designed with coprocessors in mind, it will be possible to readily add them to systems to realize those unanticipated functions without making the systems obsolete.

Fifth, even with an expanding technology, large chip sizes and smaller geometries often result in low fabrication yields, increasing the per-unit cost. In an early study, Murphy [Murp64] relates yield to the susceptible area of a device (a) and the mean defect density of spot defects per unit area (D_0):

$$yield \approx \left\{ \frac{1 - e^{D_0 a}}{D_0 a} \right\}$$

Integrating as many functions as possible on one chip is desirable to simplify interconnection problems. Since yield declines with increased chip size, it can be more cost effective to partition functions

between multiple chips to achieve reasonable fabrication yields and rely on maturing technologies (solder-bump [Elec87] or flip-chip bonding [Brad85]) to overcome packaging and interconnection problems. This holds especially true for newer technologies, where chip size and capacity are relative small.

Sixth, power dissipation becomes a problem with larger and larger chips [Sze85]. By distributing functions across multiple chips, this problem is lessened.

Seventh, additional execution elements may provide a degree of fault tolerance. A system that is able to continue operating with a failed coprocessor, even if in a degraded mode, is usually better than outright failure.

Lastly, as VLSI design and fabrication become easier through improvements in CAD systems (such as circuit synthesis using silicon compilation), these special functions will be built in much less time with much less effort.

For these reasons and many others, we believe there are strong incentives to use coprocessors in future computer systems. To place our work in perspective, we next review some non-VLSI implementations of coprocessors currently in use.

2.4. Previous Work — Non-VLSI Coprocessors

As an interesting point of history, we tried to determine when the earliest reference to the term "coprocessor" was made, both in the literature and from those who were involved in the early days as computer designers. Although it seemed unlikely, our investigation suggests that coprocessor was first applied to the Numeric Data Processor floating-point chip developed by Intel Corporation in the late 1970's [Nave80]. This early math chip was billed as a coprocessor to the 16-bit Intel i8086 single-chip CPU introduced in 1978.

The concept of coprocessor, if not the term, has been around for a long time. Gordon Bell recalls that Digital Equipment Corporation (DEC) had a history of providing optional arithmetic units, starting with the 18-bit PDP 1 (circa 1960) which included multiply-step and divide-step circuitry to implement those arithmetic functions [Bell86]. He thought the term might have been used during the early stages of development of the PDP 11/45 (circa 1972) and eventually the PDP 11/70 with its floating-point arithmetic processor. Bell currently refers to such devices, however, as *co-execution units*, and dislikes the term coprocessor applied to anything that does not fetch its own instructions.

Another early computer designer, Wesley Clark, suggested to DEC in the 1960's that a PDP-8 be used as a coprocessor to the LINC-8 system [Clar86b]. LINC was an early 12-bit machine designed to accept analog as well as digital inputs directly from experiments — hence the name Laboratory INstrument Computer [Bell78]. The PDP-8 ran some of the higher-level parts of the LINC tape logic. Both processors operated concurrently, executing their own respective instruction sets.

One of the earliest implementations of a peripheral processor concept was in the CDC 6600, begun in 1960 [Thor77]. The 10 peripheral and control processors (PPUs) shared one high-speed central processor. Each PPU received one 100-nanosecond minor cycle slot each 1000-nanosecond major cycle of the central processor. The idea of using concurrency to bury housekeeping operations during other computation was exploited with this architecture.

Next, we will consider three examples of early coprocessors that have made substantial improvements in computer performance: the attached processor, floating-point accelerator, and channel or direct memory access devices.

2.4.1. Attached Processors

In the world of scientific computing, a variety of devices have been introduced to overcome limitations of the von Neumann architecture. *Attached Processors* (AP) are generally considered a cost effective means of speeding up a compute bound task. An AP is a single processor attached to a general host computer as a peripheral (it uses standard bus cycles for communication with the host) for the purpose of speeding up a certain class of computations. It typically has a large autonomous main storage, performs floating-point at the rate of a supercomputer, and may or may not be capable of vector

operations. Examples of attached processors are the Floating Point Systems FPS 164/264 [Char81, Char86] and the Computer Signal Processing Inc. MAP 200 [Cohl81]. Certain classes of problems require such long production run times that they fall outside the realm of these devices and require the speed and efficiency of vector supercomputers, such as the Cray or CDC Cyber 205 [Thei83]. Such systems will not be considered here. Table 2-1 summarizes many characteristics of some commercially available attached processors.

Table 2-1. Commercial Attached Processors

Manufacturer Name	Model Number	Price Range L/M/H	Host CPUs	Instr Cycle (nsec)	Word Size (bits)	Number of		Micro Instr (bits)	1024 Point FFT Time (msec)
						Add	Mul		
Analogic	AP500	L	Many	160	32	1	1	N/A	4.7
CSPI	CP-232	M	Many	200	32	2	2	N/A	4.5
Data General	AP/130	M	DG Eclipse	200	64	1	1	132	8.8
FPS	164	M	Many	167	64	1	1	64	4.7
FPS	264	H	Many	53	64	1	1	64	1.6
IBM	3838	H	IBM-370	100	32	4	4	68	3.0
Numerix	432	M	DEC Only	100	32	2	1	128	1.6
Numerix	332	M	DEC Only	125	32	2	1	128	2.0
Numerix	464	M	DEC Only	125	32 or 64	2	1	128	4.0
Alliant*	FX80	H	Integrated	85	64	8	8	56	1.5

This table lists several commercial attached processors. Some are special purpose accelerators compatible only with a certain machine or manufacturers family of machines. Others are general purpose, accepting many different hosts. All are microcoded, supporting either single- or double-precision operands. Clock rates vary by a factor of 3.8, processing rates vary by a factor of 5.8. The cost designations L, M, and H are roughly: L < $20K; L < M < $200K; H > $200K. (Adapted from [Karp81] and recent product literature.) *The Alliant is an example of an integrated system with microprocessor-based interactive front-end and back-end processors for vector floating-point computation.

It is difficult to state a general conclusion about the speedup effectiveness of attached processors, since it is always application dependent. Some university researchers [Chab85] investigated the use of the MAP 200 array processor [Cohl81] attached to an HP21MXE for speech analysis. They found the communication overhead for using the attached processor was 100 times *greater* than the floating-point computation time within the attached processor. However, this was still 10 times faster than using the HP21MXE without the attached processor.

On the other hand, computations that take hours in execution may be affected little by data transfer and control overhead. Researchers at Cornell University report that attached processor time for different simulations in theoretical astrophysics and general relativity can range between 15 minutes and more than 20 hours using an FPS 190L attached processor [Faro83]. The data transfer overhead is negligible in comparison: between 0.1% and 0.001% of the total time.

Cole applied the FPS 164 attached processor to the problem of circuit simulation using SPICE and a variety of input benchmarks [Cole83, Cole85]. Compared to computation on a VAX 11/780 with floating-point accelerator hardware running 4.1 BSD UNIX, speedups ranged between three (for standard software and compiler implementations) and 10 (tuned, vectorized and optimized code). He indicated that communication overhead can account for as much as one-third of the total processing time. Careful attention to the coding of certain inner loop modules on the attached processor was found to be the most effective way to improve the execution time.

Besides the numerical processing power of the attached processor (AP), it is important to consider the control flow of the CPU-AP configuration. Two possibilities exist: (1) run the application on the host with calls to the AP for service, or (2) run the application on the AP by itself. The second alternative assumes an AP that is really a complete computer. This is not usually the case, and will not be discussed here. (Table 2-1 includes one such system, the Alliant, for comparison.) The first case presumes

a software development system for the host that at a minimum allows procedure call specification in the application program. Low level device driver software must deal with the issue of providing the AP with the data (or address of where to get it, if the AP has its own disk control) as well as other control information. It then must perform a system call. The operating system must suspend the user job, verify all addresses for validity, page-in any missing pages (for a virtual system) from disk, mark the space as *I/O active*, initiate the channel program (described later), receive a completion/error interrupt at the end of processing, release the user program as *I/O complete*, so it can be run, swapped, and so on. Thus, the complexity of the CPU-AP interaction can result in many overhead cycles during a computation. Such interactions typically consume 50 to 100 milliseconds. The AP could easily have completed 200,000 to 500,000 operations during that time.

In summary, attached processors are used in a variety of computationally intensive applications. Processing speed improvement is highly dependent on application, data sizes, and algorithm implementation. Sometimes CPU-AP overhead dominates computation time and in other cases it is completely irrelevant. System support software must be finely tuned to provide the most efficient data transfer and AP control for many applications.

2.4.2. Floating-point Accelerators

Another type of coprocessor is the floating-point accelerator (FPA), available as an option on most mini- and super-minicomputers and many workstations. An FPA is typically one or more boards that plug into a standard minicomputer backplane bus. Usually no memory is provided other than for operand and result storage, and the device is very *closely coupled* with the host CPU during operation; that is, it functions on a *single-operation-per-interaction basis* with the CPU. For example, see [Digi76, Digi81, Tayl83]. It also provides a speedup for floating-point arithmetic, but is not intended to support the same types of computation addressed by the AP. An FPA deals on an item-by-item basis with the CPU, does not support vector operations, and requires no low-level device drivers or special system calls. In fact, the operation of the FPA presumes no special software or compilers. A floating-point instruction is either executed in software (resulting from a trap, if an FPA is not present), or in the FPA. From a programmers point of view, nothing changes but performance. Typically, the FPA implements addition, subtraction, multiplication, and division directly. Other functions may include square-root, sine, cosine, and so forth.

In terms of control, if the CPU is equipped with an FPA, the initiation of an operation is automatic. Either the FPA monitors the instruction bus and recognizes instructions it is to execute when they are fetched, or it is sent a *start* signal from the CPU to initiate its operation. The first model is often referred to as an *instruction tracker*. The CPU knows it has an FPA to use either by explicit interface signals or an internal state bit. If the FPA is not present, the CPU will automatically jump to numeric microcode or call a runtime software routine.

During instruction execution, if the FPA *shadows* the CPU registers (as does the VAX 11/780 FPA [Digi81, Patt85]), the operands are already in the registers specified by the instruction or will be placed there after the effective address is calculated, and the computation proceeds. If shadowing is not provided, the operands must be explicitly transferred to the FPA registers from memory or from CPU registers. Since the CPU and FPA deal on an item-by-item basis, the CPU normally pauses until the FPA finishes an operation, allowing no concurrent execution. Traditionally, an FPA is designed to reside in a specific host CPU and is not interchangeable with other manufacturer's FPA's. The price of an FPA ranges between $8,500 and $14,000 for VAX 11 computers and represents about 10% of the cost of the rest of the system. Table 2-2 lists the execution times for various instructions for the VAX 8600, showing performance in comparison to the same instructions implemented in microcode.

In summary, the FPA requires minimal explicit control from the CPU. The FPA typically appears as a set of registers in the host CPU memory map. The existence of the FPA is transparent to the compiler and simplifies the software development problem since there is no distinction at the instruction set level between software execution and hardware execution. Although the FPA may provide an important performance improvement over run-time library implementations or microcoded routines, there is

Table 2-2. Execution Time - VAX 8860 Microcode vs. Floating-point Accelerator				
Floating Point Instruction	Operand Precision	Microcode Exec Time Cycles	FPA Exec Time Cycles	Speedup
ADDF2	Single	21	4	5.3
SUBD3	Double	44	7	6.3
MULD3	Double	60	17	3.5
DIVD3	Double	161	67	2.4

This table illustrates the speed up possible with a floating-point accelerator for various CPU's and FPA implementations. On DEC VAX 11/780, floating-point operations are 2.4 to 6.3 times faster in the FPA than microcode. Data was supplied by [Foss88, Melv88].

still a significant amount of time spent in various overhead, data transfer, and waiting functions. Also, unless the compiler allocates registers for floating-point variables, large amounts of time will be spent in transferring operands, reducing the FPA's effectiveness.

2.4.3. Channels and Direct Memory Access Devices

Besides floating-point computation, devices have been added to computer systems to improve input/output performance. If required to service a high speed peripheral, the CPU can become bogged down with nothing but I/O. For example, a disk memory can easily transfer many million bytes per second requiring a large part of the resources of a typical mini-computer or workstation. A *data channel* is a single processor attached to a general host computer as a front or back end to offload specific I/O applications from the host. Terminal interaction, print spooling, auxiliary storage transfers, and device communication are examples. Data channels can be implemented in microcode in the main CPU, but normally they are physically separate, have a limited amount of storage, a limited instruction set, and usually compete with the main processor for bus cycles and memory access. Examples include the IBM selector and block multiplexer channels [Brow72, Kuck78]. A channel is essentially a limited function programmable computer. In specific instances, these are also referred to as *direct memory access* (DMA) devices. Instead of a general purpose I/O controller, a DMA device can simply be a hardwired implementation of a single function, often referred to as a DMA *device controller*. The DMA control logic and device controller are typically located on the same board.

The main characteristic of a DMA I/O device is conveyed by its name: virtually all interaction between the device and the computer system is done through the host's memory directly, without intervention by the CPU. DMA channels are also used when there are many slow speed devices such as terminals or printers, since the collective interference to the CPU can be substantial.

To initiate a disk data transfer, a DMA device controller is given a starting address in memory, the length of the transfer, the direction (read or write), and a start signal. Once initiated, both the CPU and DMA device operate concurrently. Transferring disk data requires immediate attention, and the device signals the CPU when it needs the memory bus. This does not cause the CPU to save its state but only relinquish control of the bus. Logically, the DMA controller looks like a set of registers in the memory map of the host. Programmed I/O transfers to those registers are used to initialize, start, and check status of the device.

DMA channels are slightly different. For example, the IBM channel is implemented as a limited function, free standing computer. Its programs, data, and control reside in the main CPU memory. Channel control programs are initiated by the CPU when it encounters an I/O instruction, and from that point on, the channel competes with the main processor for bus cycles for its instructions and data transfers. A successful initiation of a channel request results in the channel reading a channel control word (CCW) from main memory at a fixed location. This word provides the storage protection key for the transaction, first command address, device status bits, and the next CCW address. I/O completion is signaled by an interrupt to the CPU by the channel [Blaa64, Siew82].

In summary, the use of specialized DMA and channel computers presumes that all the overhead associated with their use is amortized over the period of action. Only a few instructions by the CPU are needed to effect the transfer of vast amounts of data that would otherwise swamp the CPU if it had to deal with it directly. Independent execution allows the CPU and DMA devices to operate concurrently, although memory bus contention will occur.

To summarize this section, we note that despite the obvious parallelism inherent in having two independent execution elements, coprocessor applications are often still characterized by serial processing. In many cases, communication between the devices diminishes much of the potential performance advantage gained by having the special hardware assistance.

2.5. VLSI Coprocessors

Besides the central processor, many VLSI components have augmented microprocessor-based systems. In this section, we briefly survey what has been done to improve computer systems by the use of additional VLSI components tailored to specific tasks: floating-point arithmetic, memory management, graphics, text processing, and so forth. Some of these will be considered in much greater detail in later chapters.

2.5.1. Floating-point Arithmetic

A floating-point coprocessing unit is one of the many applications made possible by advances in VLSI. Some of the commercial floating-point coprocessors designed for use with microprocessor-based systems are shown in Table 2-3.

Table 2-3. Commercial VLSI Floating-point Coprocessors

Part Number	Manufacturer	Approx # Xstrs	Technology Used	Speed (Mhz)	# Pins for Data	Arithmetic		FADD (μsec)	FMUL (μsec)
						Basic	Other		
78132	DEC	200,000	3.0 μ ZMOS	5	32	+,-,x,÷,≡	int x,÷,poly	3.8	3.0
i8087	Intel	65,000	3.0 μ HMOS	10	16	+,-,x,÷,≡	√,trig	14.0-20.0	18.0-29.0
i80287	Intel	65,000	3.0 μ HMOS	16	16	+,-,x,÷,≡	√,trig	8.8-12.6	11.3-18.1
i80387	Intel	75,000	1.25 μ CHMOS	16	32	+,-,x,÷,≡	√,trig	2.9-3.9	3.6-7.1
MC68881	Motorola	155,000	2.0 μ CMOS	16.7	32	+,-,x,÷,≡	√,trig	3.1	4.3
MC68882	Motorola	165,000	1.5 μ CMOS	16.7	32	+,-,x,÷,≡	√,trig	3.4	4.6
NS32081	National	55,000	3.0 μ XMOS	10	16	+,-,x,÷,≡	none	7.4	6.2
R3010	MIPS	75,000	1.6 μ CMOS	25	32	+,-,x,÷,≡	none	.08	.16-.2

This table provides a brief summary of the capabilities of some commercial floating-point coprocessors. An extensive summary of performance, interface characteristics, operations supported, and so forth is in [Fand85]. These implementations support from two to eight different data types. All coprocessors maintain eight 80-bit user-accessible data registers on chip except the microVAX 78132, which has no user-accessible registers, and the R3010 which has 16. The FADD and FMUL columns show typical execution times for double precision (64-bit) register-register operations for the cycle-time given. Most coprocessors are available in a variety of clock speeds between 6 MHz and 25 MHz. The R3010 is the most recent addition and reflects the latest technology and simplified architecture, providing quick fundamental operations [Digi85, Inte81a, Inte85a, Inte85b, Moto85, Nati84, Pren87, Rowe88].

The floating-point chips listed in Table 2-3 span nearly a decade in time from the introduction of the Intel i8087 numeric data processor to the MIPS Computer Systems R3010 floating-point coprocessor. Consequently, a wide range of performance is seen. Nevertheless, there are some fundamental design ideas and interface strategies that generally affect performance and effectiveness. Chapter 4 considers this in more detail.

2.5.2. Memory Management

For both cost and performance reasons, VLSI microprocessors are well established as the central building blocks in most contemporary general-purpose and workstation computer designs. Recently introduced high-performance personal super computers rely on off-the-shelf CPU chips (for example, see [Mola88] for discussions of the Stellar and Ardent systems.). These hardware configurations are required to run general-purpose time-shared operating systems. Performance in dealing with the register-cache-main-memory hierarchy and efficiently implementing demand paging for the virtual memory system go beyond the capabilities of most one-chip CPU's. Consequently, a memory management unit (MMU) is an absolute necessity in contemporary systems. Without the MMU, the CPU is severly constrained in either memory (i.e., small memory size, directly addressable and non-cached), or performance (i.e., the CPU attempts to handle memory management functions).

Most microprocessor manufacturers provide MMU's compatible with their CPU's. The Motorola MMU for the MC68000 family of microprocessors is one of two predefined tightly-coupled coprocessors supported by the generalized coprocessor interface. The cache controller for the Titan system, a research multiprocessor developed by DEC [Joup88], is also viewed by the CPU as a tightly-coupled coprocessor. The interested reader is referred to [Wils88a] which lists MMU's and other support chips and functional blocks to augment most microprocessor systems.

2.5.3. Other VLSI Coprocessors

As a means of improving the performance of other computer system functions, specialized devices have been added from time to time. Depending on speed, functionality, and a number of other factors, these are realized as either multiple-chip building block components or monolithic single-chip implementations.

There are also several groups pursuing research in the area of coprocessor architectures, including an effort to develop a "design frame" [Borr85] approach for the Motorola 68000 family [Chat87]. Many commercial devices are cataloged in trade publications (see [Hear88]). Some of these, and ongoing research, include:

- serial and parallel I/O controllers,
- device controllers (Winchester or SCSI disks, terminals, and so forth),
- digital signal processors (single chip microprogrammable devices or building blocks),
- graphics processors (monolithic coprocessors or chip-sets to do *bitblt*, raster control, data shifting, clock generation, video D/A conversion, and so forth),
- communications adapters,
- performance monitor coprocessor [Fauc86],
- text and language coprocessors [Kung81],
- string manipulation coprocessor [Curr83],
- persistent object coprocessor [Geor87],
- database coprocessor [Anon85],
- network controller coprocessors (conflict resolution/arbitration),
- sorters [Care85, Mira83],

and many others.

2.6. Chapter Summary

This chapter has surveyed past and present developments in the area of coprocessor functionality. The prime motivations for developing specialized hardware are performance improvement and system balance. From a control point of view, coprocessors span the spectrum of highly visible peripheral-like devices — attached processors to digital signal processor chips — to unseen yet performance-critical devices — from memory-management units to *bitblt* chips. In between are devices that the high-level language programmer becomes aware of only in terms of performance — from the floating-point accelerator board to the floating-point coprocessor chip.

From an interface point of view, coprocessors again accommodate a range of styles — from closely coupled synchronous devices to loosely-coupled asynchronous peripheral devices. The closeness of the interface is usually proportional to how time-critical the coprocessor function is to overall system performance. As the ratio of the number of CPU cycles needed to interact with the device (setting it up, providing data, interrogating results) compared to the number of cycles expended by the coprocessor in doing its function (multiply two floating-point numbers) get closer to 1.0, the more closely the coprocessor must interact with the CPU to guarantee its usefulness.

In Chapter 3, we identify categories, types, and styles of coprocessors and develop a performance model that accounts for features of the complete computer system that influence coprocessor effectiveness and utilization.

<This page is intentionally blank.>

3 Coprocessor Classifications and Performance Analysis

3.1. Introduction and Overview

In Chapter 2, we reviewed some past and contemporary coprocessor implementations. Coprocessors can be as varied as the applications they serve, and consequently, there are many ways to classify them. The first and possibly most logical is along functional lines; i.e., math coprocessors, graphics coprocessors, string or text coprocessors, and so on. Within a function it is possible to separate them into general-purpose versus special-purpose implementations. At a lower level, hardware protocols may categorize an implementation as closely-coupled or loosely-coupled.

Nevertheless, this chapter identifies some common characteristics of coprocessors and seeks to categorize them and identify types or styles of coprocessor architectures. Once classes are identified, common performance parameters are established to help analyze and evaluate coprocessor effectiveness. The goal of this study is to discover what things contribute to making a coprocessor effective, and let that knowledge guide the design of future coprocessor architectures. Sections 3.2 and 3.3 briefly consider software and hardware issues related to the incorporation of a coprocessor into a computing environment. Section 3.4 presents a model for evaluating coprocessor effectiveness in the context of its system, and identifies the essential parameters influencing the overall system performance related to the coprocessor. Subsequent chapters show the application of this model in the evaluation of some VLSI coprocessors.

3.2. Software Issues

Since we have defined the coprocessor as a device that replaces software routines, one of the primary considerations would be the integration of the coprocessor with the rest of the software. In what form does the coprocessor make its presence known in the software? Is the coprocessor seen in the instruction opcode, or at the function level, or at all? Is the coprocessor invoked through I/O instructions, or is it independent of the user program and receive direction from the operating system through supervisor calls or hardwired signals? These are all determined by the nature of the various

computational forms and depend on whether the coprocessor assists at the instruction level, the function level (e.g., a subroutine), or the entire algorithm level (e.g., a program).

To determine that a coprocessor is necessary to accomplish the performance goal of a system involves a thoughtful investigation. It is very important to realize that the potential benefit must be well considered in light of the cost. This would seem like a foregone conclusion, but there are many instances where coprocessor accelerators for specific tasks have become function decelerators for a program in general. For example, the redisplay of the mask geometries using an interactive graphics editor for VLSI can be a very time consuming process (i.e., consumes many ten's of seconds up to minutes). This is frustrating to the designer and a waste of his time. An early version of a graphics accelerator for a workstation environment used for VLSI design was shown to actually *slow down* the redisplay function [Mayo86]. Consequently, it was not used. In another case, a coprocessor designed to assist in the BitBlt* operations of a monochrome workstation display was found to interact poorly with the rest of the system (i.e., conflict with the CPU made it ineffective), added nothing in terms of performance, and yet was a major problem for system reliability [Bizj86]. Again, the coprocessor was not useful and was designed out of later versions.

There is no magic formula for determining the proper split between special purpose software and special purpose hardware. If it is possible to identify a particular computation and the companion data structure upon which the computation will be performed, then a specific analysis can be undertaken to determine the benefit. Section 3.4 considers this and related topics in more detail.

3.2.1. Software Interface

As an example of a coprocessor that assists at the instruction level, consider floating-point arithmetic. Floating-point instructions normally require two unique operands, generating a change to one operand or possibly producing a third unique result. Since instructions in many machines consume only a single clock cycle, the interaction with a coprocessor to assist the floating-point computation must be quick and efficient.

As an example of a coprocessor at a function or subroutine call level, consider image processing. Often it is necessary to compute the convolution of an image with some filter. For each pixel, its value designated as $V(x,y)$, the Gaussian convolution of radius r is given in equation 3-1

$$V'(x,y) = \sum_{i=-r}^{r} \sum_{u=-r}^{r} C_{i,j} V(x+i, y+j) \qquad (3\text{-}1)$$

This computes V', a new value for V. It is essentially a blurring step to eliminate visual noise of frequency less than distance r. Given the values for x, y, i, j, and r, the coprocessor could compute the pixel value V' without any other interaction or direction from the host.

As an example of a coprocessor that implements a complete program, consider a complete image processing system. In addition to the filtering function mentioned above, a complete image processing system must include functions for contrast enhancement, noise rejection, edge extraction, edge enhancement, and various transformations [Ruet86]. The full system might have coprocessors for each of these functions.

3.2.2. Compiler Issues

For coprocessors that are seen at the instruction level, compiler issues become significant. First, the compiler needs to understand and recognize the execution model of the coprocessor. If the coprocessor consumes several cycles for its basic operations, a compiler optimization to intersperse instances of coprocessor commands with other non data-dependent and non-related code allows the CPU to execute concurrently. Without optimization, significant performance improvements may be lost. As a

*BitBlt is a contraction for *bit block transfer*, and is the process of moving and modifying regions of bits in bit-mapped graphics displays from visible portions of the screen to "invisible" regions (in memory), and vice-versa. The technique was developed at Xerox PARC. It is also called "RasterOp".

means of accommodating a first-order level of concurrency, the hardware protocol may allow the CPU to proceed with other instructions after the issuance and before the completion of coprocessor instructions. To the extent that sequential instructions are independent or do not require the coprocessor resources, parallel execution can occur.

For simplicity, the best of all possible worlds would allow the inclusion of a coprocessor in a software system and yet not require the compiler be modified at all to accommodate it. That is unlikely if performance is a consideration. The simplicity of the programming model of the coprocessor can simplify that task, however. This may portend compiler generators if new compilers are needed. For a generalized coprocessor, the *design frame* approach [Borr85] provides general coprocessor instructions at the instruction set level and it is up to the individual coprocessor along with certain restrictions based on address or other identifiers to interpret the commands appropriately. The Motorola 68000 family provides a general coprocessor interface supporting up to eight different coprocessors, although only two coprocessors — the memory management unit and floating-point unit — are directly supported in the instruction set at this time [Cass84, Moto87].

Coprocessor operations can be controlled either at compile time or run-time. Even for systems equipped with a hardware coprocessor, it is often a user-controlled feature of compilation to include explicit coprocessor instructions in the code generated. Without the instructions, library routines are linked and parameters are passed to the coprocessor through memory, the user-process stack. With explicit coprocessor instructions, transfers and operations are defined functions of the hardware and operands may be transferred from registers, memory, or reused in the coprocessor. For some systems, if the coprocessor is absent or disabled, executing the instruction will result in a trap to routines that implement the function.

3.2.3. Operating System Issues

Many of the complex issues that arise in computer systems software come from the various abnormal events that can occur during the course of execution. These are called faults or exceptions, and are often accompanied by interruption of the normal flow of control. A coprocessor that can generate or cause any type of exceptional condition implies the need of the operating system to handle that exceptional condition. If a coprocessor is seen as a device to the operating system, a device driver is needed that contains exception handling routines. If the coprocessor is transparent to the operating system, errors may go undetected. Whether that causes a problem depends on each situation.

In the case of memory management, the amount of data either required or produced by the coprocessor must be known to the operating system. An interrupt per datum is fine for quick-response to a terminal keystroke, but completely inappropriate for data-block transfers to a graphics device. The data requirements based on both volume and rate and statistics on page size, block size, and operand size determine the best fit of the coprocessor within the software environment.

3.3. Hardware Issues

As with software, there are many hardware characteristics that differentiate coprocessors. Broadly speaking, there are six classifications, depending on various control and data-related factors. These include:

- the instruction or command paradigm,
- the data transfer protocol,
- the data types supported,
- the memory hierarchy interaction,
- the interconnection topology, and
- performance and speed.

There is a correlation between the physical placement of the coprocessor within the computer system hierarchy and these hardware issues. To guide our discussion in sections that follow, Figure 3-1 illustrates three levels of coprocessor hierarchy within a computer system.

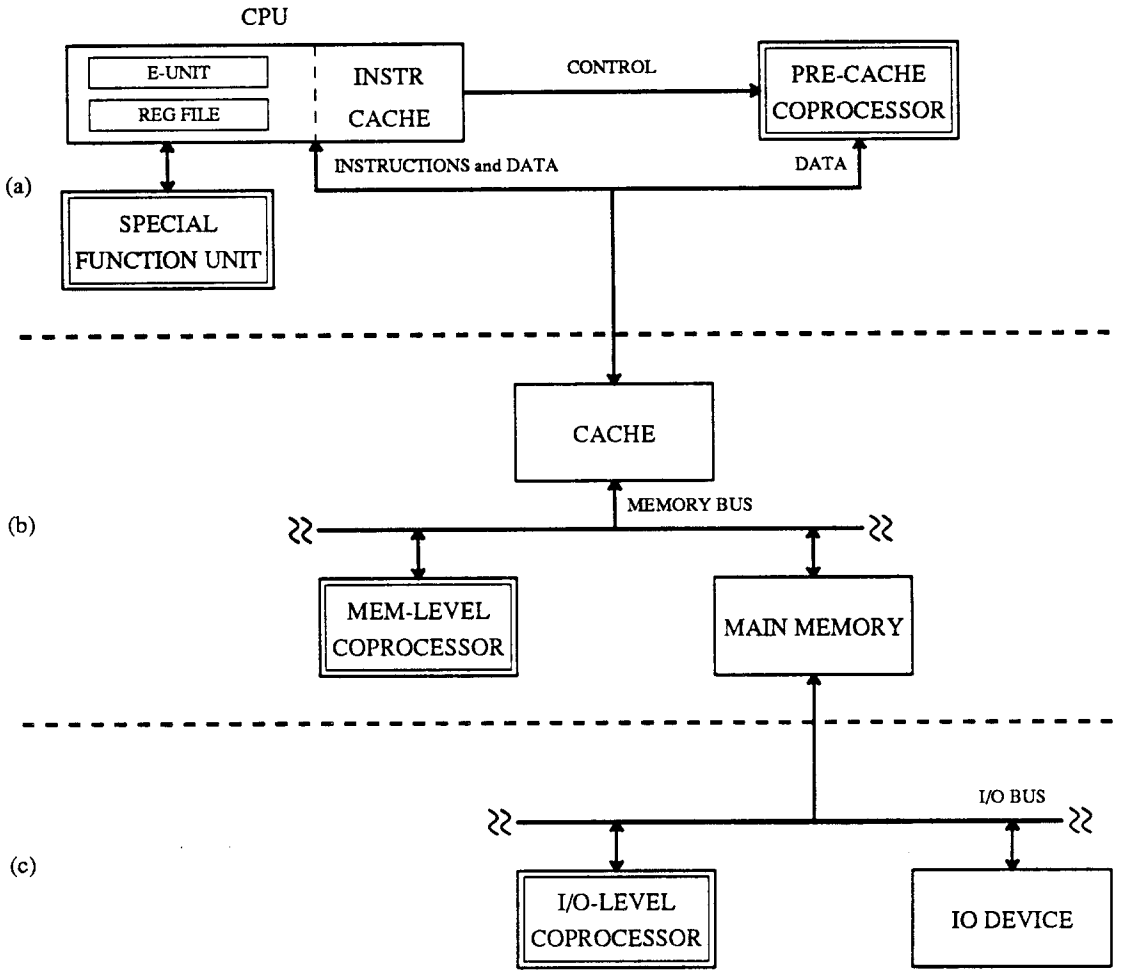


Figure 3-1. Topological Levels of Coprocessor Interconnection.

This figure shows various points in a computer systems hierarchy where coprocessors could be attached. Such things as instruction-fetch paradigm and data transfer requirements need to be considered to determine the most effective configuration for a specific function. Part (a) illustrates CPU-level coprocessors; Part (b), Memory-level coprocessors; and Part (c), Input/Output-level coprocessors.

On the basis of the interconnection topology, coprocessors are grouped in three levels:

- CPU-level coprocessors, including special function units and pre-cache coprocessors,
- Memory-level coprocessors, and
- Input/Output-level coprocessors

At each of the three levels, there are two main things to consider:

- instruction sequencing and/or control, and
- data/operand manipulation.

In the following sections, we consider inter-level differences and similarities and intra-level properties for control and data for each of the three levels and relate them to the six classifications above.

3.3.1. Instruction and Control Issues

According to Gordon Bell, the term coprocessor should be used only with those devices that can fetch and execute their own instructions [Bell86]. He calls a device that simply receives commands from the CPU a "co-execution element." Both of these are examples of CPU-level coprocessors. Some devices have built-in or hardwired instruction sequences and allow no generalization as provided by an instruction stream. Such coprocessors can exist at any level.

For coprocessors that do operate off the instruction stream, there are several ways of providing control. The instruction could be:

- fetched by the CPU and seen simultaneously by the coprocessor, or
- fetched by the CPU and sent to the coprocessor, either in encoded or decode form, or
- fetched by the coprocessor itself.

We call the first of these an *instruction tracker* coprocessor. As the name implies, the coprocessor tracks or follows the instruction stream as it comes out of storage and decodes and executes those instructions intended for it. The CPU execution unit essentially treats coprocessor instructions as "no-ops." These are either CPU-level coprocessors or memory-level coprocessors and interact directly with CPU instruction buffers, off-chip instruction caches, and main memory. If the CPU maintains an on-chip instruction cache, the instruction stream must be observed via a special control path from the CPU, as shown in Figure 3-1a. Some floating-point coprocessors, such as the Intel i8087, and high-performance graphics engines are examples of this type of device [Harr85, Nave80]. I/O-level coprocessors do not use instruction-tracking protocols.

We refer to the second instruction-issue method as a *master/slave* protocol. The CPU is the master and the coprocessor is the slave, and receives direction and begins operation only on command by the CPU. These are usually memory-level coprocessors, but in some cases are found at the CPU-level and I/O-level. This method is less tightly-coupled than instruction-trackers and is used for general purpose coprocessors that may operate asynchronously with the CPU and do not interact as closely with the CPU pipeline, for example. The distinction between CPU-level and memory-level coprocessors is not very clear when the system does not have a cache memory. The Intel i80287 and Motorola MC68881 floating-point coprocessors are examples of this instruction issue style. Many I/O level coprocessors receive some portion of their control in a master/slave manner. A digital signal processor chip is another example of a master/slave coprocessor [Wils88a, Wils88b].

The third instruction issue paradigm we refer to as *autonomous*, since the coprocessor has the ability to control its own instruction stream or continue execution under its own control. These types of coprocessors are nearly always found at the I/O-level of the interconnection hierarchy. The peripheral processing units of the CDC 6600 were autonomous coprocessors to the high-speed central arithmetic unit. A DMA controller is also an example of an autonomous coprocessor. In a multiprocessor system, a second CPU is used to implement certain aspects of an algorithm may be considered an autonomous coprocessor, but that has more to do with software execution than hardware issues and will not be considered further.

In many cases, coprocessor control units are combinations of all three methods. One part of the CPU-to-coprocessor protocol may be master/slave followed by autonomous action by the coprocessor as it continues to fetch and execute its own instructions until finished. We next consider aspects of data transfer and manipulation in coprocessors.

3.3.2. Data Transfer and Memory Interaction

Besides the flow of instructions to the coprocessor control unit, operands must also be provided. There are at least five ways that coprocessors send or receive data:

- the coprocessor has "built-in" hardwired constant values,
- the CPU executes load/store instructions, but the data are received/sent by the coprocessor,
- the CPU transfers data to its internal registers, then writes to the coprocessor,
- the coprocessor executes its own load/store and transfer operations,
- the coprocessor has registers that shadow the CPU registers (that is, data that are used by the coprocessor appear in both CPU and coprocessor registers due to a special connection with the CPU).

At the CPU-level, special function units extend the CPU datapath and operate on the same data types. Consequently, they are synchronous with CPU operations. They may have their own registers or use CPU registers for high-speed accesses. On- and off-chip caches provide fast access to operands involved in such specialized functions as fixed-point binary arithmetic, array index manipulations,

emulation functions, encryption tasks, and so forth [Birn85, Ples86]. Although implementations can be pipelined to minimize the penalty for chip-crossings, off-chip memory is usually slower than registers. Special function units that interact directly with CPU registers need to be carefully designed to avoid slowing the entire cycle-time of the system. The data manipulated are usually scalar-like, rarely involving large blocks.

Pre-cache or memory-level coprocessors often manipulate data types that are different from those used by the CPU or special function units. Consequently, they may maintain separate register files. They receive/send data by monitoring the data bus or respond to move operations between it and CPU registers. The data involved are typically individual words instead of large blocks. Since the data types may be different than those used by the CPU, the bandwidth between pre-cache or memory-level coprocessors and storage may be quite different from the CPU. For example, double-precision floating-point operands may pass directly between floating-point coprocessor registers and the data cache in a single cycle, while CPU accesses may require multiple-cycles to transfer the same data.

I/O-level coprocessors typically transfer large blocks of data between devices and main memory with little or no interaction with the CPU. The interaction with main memory can either be cycle-stealing or uninterruptible burst transfers. The model for data manipulation matches that of control — very little interaction with the system once initiated.

Having considered many of the issues that distinguish and separate different types of coprocessors at the interface level, and ways in which coprocessors interact with the rest of the system to send, receive, and manipulate data, we next develop a performance model for coprocessors that is a function of both instruction issue and data manipulation paradigms.

3.4. Coprocessor Performance Analysis Model

Since the coprocessor is intended to be a means of speeding up certain software functions with special purpose hardware, there should a one-to-one correspondence between the software algorithms and the hardware devices that replace them. To justify the inclusion of special purpose hardware in a computer system, it is important to determine the potential speedup that comes from adding the coprocessor.

If the time to perform some computation using algorithm A with only software routines on a uniprocessor is designated $T_{SWO}(A)$, then the time to do the same task using a coprocessor to replace *some parts* of A can be designated $T_{SW+HW}(A)$. The *speedup* achieved is simply the ratio of the two execution times:

$$Speedup(A) = \frac{T_{SWO}(A)}{T_{SW+HW}(A)} \tag{3-2}$$

Many factors must be considered to determine if the speedup justifies the inclusion of the coprocessor in the specific computer system using it for application A. Of interest to us is the term $T_{SW+HW}(A)$ and what it implies for the use of coprocessors in computer systems in general and the implied cost effectiveness of coprocessors in specific cases.

If the fundamental structure of task A does not change substantially with the use of a coprocessor, then those sections of A that must be run on the general purpose host can be identified and separated from those using the coprocessor. In this case we can designate the portion of the algorithm that must be run on the host as A_{SW} and the part that can be assisted by the coprocessor as A_{HW}. Thus, the time to run the application using only software is $T_{SWO}(A) = T_{SW}(A_{SW}) + T_{SW}(A_{HW})$, and a first order approximation of the time to complete A using both host software and coprocessor hardware (assuming a serial-execution model) becomes This is analogous to Amdahl's law [Amda67],* which suggests that there are fundamental limits to the application of parallelism (in this case, a host CPU plus coprocessor) in achieving performance improvement.

*Amdahl's law, as expressed in [Gust88], for an N processor multiprocessor system: If s represents the time spent by a serial processor on the serial parts of a program, and p represents the time spent by a serial proces-

$$T_{SW+HW}(A) = T_{SW}(A_{SW}) + T_{HW}(A_{HW}) \tag{3-3}$$

With this assumption, the speedup possible due to the addition of a coprocessor is

$$Speedup(A) = \frac{T_{SWO}(A)}{T_{SW}(A_{SW}) + T_{HW}(A_{HW})} \tag{3-4}$$

$$Speedup(A) \lim_{T_{HW}(A_{HW})\to 0} \leq \frac{T_{SWO}(A)}{T_{SW}(A_{SW})} \tag{3-4a}$$

This analysis is based on the model that the execution of application A is strictly sequential. That holds true for the software-only implementation, where the computation begins at t_0 and ends at some later time, t_z, with $T_{SWO}(A) = t_z - t_0$. In the second instance, the same application A with a host and coprocessor, the computation begins at t_0 and ends at t_v, such that $T_{SW+HW}(A) = t_v - t_0$. The results should be identical with the only change evident to the programmer being the difference in time, hopefully $t_v \leq t_z$. However, besides the fact that $T_{HW}(A_{HW}) \leq T_{SW}(A_{SW}) + T_{HW}(A_{HW}) \leq T_{SWO}(A)$, is the fact that both A_{SW} and A_{HW} can be computed simultaneously, in which case $T_{SW+HW}(A) \leq T_{SW}(A_{SW}) + T_{HW}(A_{HW})$. In other words, the time to execute A on a coprocessor can be *less* than the sum of the times to execute the separate pieces in SW and HW respectively. In this case, the time $T_{HW}(A_{HW})$ does not have to be zero to achieve maximum theoretical speedup.

A second effect to consider is pointed out in recent research suggesting that the assumptions underlying Amdahl's 1967 argument may not hold, due to the fact that the structure of A can change when using multiple execution elements [Gust88]. Gustafson cites examples of massive parallelism where actual speed up is several factors greater than predicted by Amdahl's law. How does this relate to coprocessor applications? Besides the fact that $T_{HW}(A_{HW}) \leq T_{SWO}(A_{HW})$, requiring less execution time than software, and that $T_{HW}(A_{HW})$ can execute concurrently with $T_{SW}(A_{SW})$, the structure of A often *changes* (to, say, (A')) with the inclusion of additional execution elements. For example, with more processor power to address the problem, the problem domain is likely to expand so the time spent in computation remains constant, rather than reduced. In other words, the problem size scales with the number of processors, and p and N are linearly related, rather than being independent as assumed by Amdahl's law. In this case, a *multiplicative* effect results with an improvement in performance:

$$T_{SW+HW}(A') \leq T_{SW}(A'_{SW}) + T_{HW}(A'_{HW}) \leq T_{SW+HW}(A) \leq T_{SW}(A_{SW}) + T_{HW}(A_{HW}) \tag{3-5}$$

It is this effect along with others that coprocessor architectures seek to exploit.

In the next section, we identify factors associated with the various computation times discussed here and provide a model of coprocessor performance which helps predict the overall effectiveness of coprocessor based applications.

3.4.1. Factors Affecting Coprocessor Performance

For the commercial VLSI coprocessors highlighted in Chapter 2, execution time is likely to be influenced by several things, but has traditionally been separated into two categories:

- the time necessary to perform operations directly on-chip in the hardware, and
- the time to perform the same operations, but with an intermediate interaction with the memory system, either before or after the operation.

For example, with a floating-point coprocessor, it is not unusual to find performance data specified for both register-to-register operations and memory-to-register operations. The difference in execution time is likely to stem from the effective address calculation and memory bus cycles needed to transfer

---

sor on parts of a program that may be computed in parallel, and letting $s + p \equiv 1$, then speedup is:

$$speedup = (s + p)/(s + p/N)$$

$$= 1/(s + p/N)$$

operands. Similarly, other coprocessor functions often require an initial *set-up* before beginning, and normal housekeeping operations during the course of the computation. We believe that those influences on performance must be kept well in mind when designing coprocessors. Conversely, coprocessor performance can be well below the design expectation if the effects of those factors are ignored to any extent.

We would like to understand the operation of the coprocessor in the system, to be able to analyze and predict its effectiveness. It is important to establish criteria to evaluate that effectiveness and be able to model the operation easily, and to verify the models with actual measurements on real coprocessors to validate the assumptions made. The best possible result would be an accurate analytical model that could be applied to coprocessor uses. We will base our model and analyses on consideration of how the various cycles in a computation using a coprocessor are spent.

We consider all cycles either *operation* cycles or *overhead* cycles in accomplishing a result. For example, cycles spent by a floating-point unit to compute a result (i.e, the floating-point unit ALU is busy) are computation cycles, while the calculation of an array address, incrementing of a counter variable, testing a value against a loop index for termination, or transferring an operand are all consider overhead cycles. By careful examination of the hardware on which the programs run as well as the instruction sequences themselves, it is possible to categorize all cycles in the course of a computation as either operation or overhead cycles.

Some of the factors to consider in this analysis of overhead include:

- the number of actual coprocessor execution cycles to perform the operation,
- the number of cycles associated with operand transfers to the coprocessor,
- the number of cycles associated with instructions exclusive of coprocessor instructions,
- the CPU cycle time,
- the coprocessor cycle time,
- the cache cycle time in terms of both hit and miss servicing,
- the width of the data path between the coprocessor and other elements in the system,
- the use of various addressing modes allowed or provided by the host CPU,
- the general CPU architecture (i.e., load/store or memory-to-memory).
- the general coprocessor architecture, including the various data manipulation mechanisms, control features, and synchronization protocols,
- the software specification of operand addresses and how they are computed, loop index calculations, loop condition test and branch, and
- the storage of temporary results.

All these factors will influence the effectiveness and ease of use of the coprocessor.

Using the timing nomenclature of the previous section, we can reformulate $T(A')$ in terms of operation and overhead cycles:

$$T_{SW+HW}(A') = T_{COMPUTATION} = T_{OPERATION} + T_{OVERHEAD} \tag{3-6}$$

and if $T_{OPERATION} \equiv$ (coprocessor ALU not idle), and $T_{OVERHEAD} \equiv$ (coprocessor ALU idle), then we can refine overhead somewhat:

$$T_{OVERHEAD} = T_{OPERATION\ OVERHEAD} + T_{PROGRAM\ OVERHEAD} + T_{MEMORY\ OVERHEAD} \tag{3-6a}$$

And, each of these factors can be defined as:

$$T_{OPERATION\ OVERHEAD} = T_{INSTR\ FETCH} + T_{DATA\ LD/ST} \tag{3-6b}$$

$$T_{PROGRAM\ OVERHEAD} = T_{LOOP\ INDEX\ ARITHMETIC} + T_{TEST/BRANCH} + T_{ADDRESS\ CALC} + T_{NO-OPS} \tag{3-6c}$$

$$T_{MEMORY\ OVERHEAD} = T_{COPROCESSOR\ STALL\ ON\ UNALIGNED\ ACCESS} + T_{COPROCESSOR\ STALL\ ON\ LOAD\ CACHE\ MISS} + T_{COPROCESSOR\ STALL\ ON\ STORE\ CACHE\ MISS} \tag{3-6d}$$

In our investigations, we have attempted to identify and quantify the various factors that affect the performance of each system, as explained above. In our subsequent analysis of real systems, we present the execution times and overhead factors in terms of *clock ticks*.* Using the number of clock ticks provides an unbiased comparison between architectures if percentage of operation for each contributing factor is computed and accounted for. It also allows previous or future generations of the same architecture using different technology to be considered since it is a simple matter to compute the real execution time for a specified technology given the clock frequency. Often, when technology provides a faster cycle time, the number of clock ticks per operation remains the same.

From the above categories of overhead and from observations we have made of the steady-state performance of some commercial systems, we believe the time consumed to execute critical portions of programs can be expressed deterministicly, with the variables mentioned above included in the Coprocessor Performance Model Equation (3-7)

$$\begin{aligned} Total\ Cycles \le & \sum_{i=1}^{n\_cp\_ops} \left(T_{fetch}(i) + T_{cp\_op}(i) + \left(T_{cp\_op}(i) - Interval(i) \right) \right) + \\ & \sum_{j=1}^{n\_cp\_ld} T_{cp\_ld}(j) + \sum_{k=1}^{n\_cp\_st} T_{cp\_st}(k) + \sum_{l=1}^{n\_cp\_data\_interlock} T_{cp\_data\_interlock}(l) + \\ & \sum_{m=1}^{n\_cp\_ld\_u} T_{cm\_ld}(m) * EMR + \sum_{n=1}^{n\_cp\_st\_u} T_{cm\_st}(n) * EMR + \\ & \sum_{o=1}^{n\_non\_cp} T_{non\_cp}(o) \end{aligned} \tag{3-7}$$

where

Total Cycles ≡ time in cycles to execute a program,

n_cp_cps ≡ number of coprocessor operations in the program,

T_{fetch} ≡ time in cycles to fetch an instruction,

*We use the manufacturers' specified clock input frequency, *N*, to define the basic clock cycle, one clock "tick". Some data books use the term "clock count," which may be more than one clock tick, and refer to the coprocessor as an *N*/2 MHz part. This creates confusion, and is avoided here.

T_{cp_op} ≡ time in cycles to complete a coprocessor operation,

Interval ≡ distance between $cp\_op\,(k)$ and instruction (k') causing execution unit interlock,

n_cp_ld ≡ number of coprocessor loads in the program,

$T_{wait\_state\_r}$ ≡ time for read-memory wait-states,

$T_{wait\_state\_w}$ ≡ time for write-memory wait-states,

$T_{cp\_ld} \equiv T_{wait\_state\_r} + (T_{transfer} * \textit{operand size/bus width})$,

n_cp_st ≡ number of coprocessor stores in the program,

$T_{cp\_st} \equiv T_{wait\_state\_w} + (T_{transfer} * \textit{operand size/bus width})$,

$n\_cp\_data\_interlock$ ≡ number of coprocessor data interlocks in the program,

$T_{cp\_data\_interlock}$ ≡ time in cycles spent waiting for loaded operands to become valid (i.e., written to the coprocessor register file),

$n\_cp\_ld\_u$ ≡ number of unique coprocessor loads in the program,

$n\_cp\_st\_u$ ≡ number of unique coprocessor stores in the program,

EMR ≡ the expected miss ratio for the computation,

T_{cm_ld} and T_{cm_st} vary with bus contention, arbitration time, bus bandwidth, and so on, and

EMR ≤ operand_size / transfer block size.

It is important to realize that the right hand side of Equation (3-7) represents an upper bound on system performance. As discussed previously, one of the enticements to using coprocessors is the concurrent execution model provided by having them in the system. With our definition of operation time, we can eliminate time-wasting overhead cycles by managing to keep the coprocessor continuously busy. If we define the time spent in application A on operation cycles as $T_{CP\ BUSY}$ and the time that the coprocessor is idle as $T_{CP\ NOT\ BUSY}$, a simple metric of how well the coprocessor is being used is defined as:

$$Utilization_{COPROCESSOR}(A) = \frac{T_{CP\ BUSY}}{T_{CP\ BUSY} + T_{CP\ NOT\ BUSY}} \tag{3-8}$$

If $T_{CP\ BUSY} + T_{CP\ NOT\ BUSY} \equiv 1.0$, then the utilization is simply the percentage of time the coprocessor is busy during the computation. As overhead is reduced to zero (i.e., $T_{CP\ NOT\ BUSY} \rightarrow 0$) and the coprocessor execution unit is kept continually busy, the $Utilization_{COPROCESSOR}$ approaches 100%. In a multiprocessor system where the coprocessor could be used by more than one CPU, U can be used to determine whether additional coprocessors are needed.

3.4.2. Some Implications of the Concurrent Execution Model

Besides the obvious speed advantage of allowing concurrency, the coprocessor provides a certain amount of redundancy to the system. This redundancy can serve to increase system fault tolerance and availability. Also, by having multiple execution units (for example, more than one FPU per CPU), the concurrent execution model allows partitioning of algorithms and functions to take advantage of the non-multiplexed execution elements. The advantages of this may be obviated by the speed of the coprocessor, if fundamental operations are relatively fast (i.e., on the order of normal CPU instructions).

Along with the advantages come certain disadvantages of the concurrent execution model. First and foremost is perhaps the indeterminate interrupt or exception, called the *imprecise interrupt* in [Ande67]. If the host CPU and coprocessor are allowed to execute in parallel, it is clear that the host will have initiated a coprocessor instruction and then gone on to other unrelated operations. At the point where certain exceptions or interrupts due to exceptions occur, the CPU will no longer be in the

same context (i.e., strict instruction sequence) of the instruction stream when the coprocessor operation was initiated. This can present a number of problems in terms of backing out instructions, suspending operations, or identifying the operation that caused the error.

Another problem occurs with context switching for a coprocessor that maintains its own state. If that state is very extensive, the amount of time necessary to save and restore it can be significant. Depending on the application and frequency of context switching, an extremely fast means of context switch may not be necessary. The possible hardware and software complications implied with a fast context switch must be evaluated in terms of overall system performance. The ability of software to perform a simple sequence of regular load/store operations may be just as effective as special context save and restore operations built into the coprocessor hardware.

The third problem, related to the first, comes from the necessity to resynchronize the operation of the host CPU and that of the coprocessor. Certain types of operations will be self-synchronizing, in the sense that one will not begin until the previous one finished. On the other hand, operations that can be initiated by the CPU but are dependent upon results produced by the coprocessor have inherent hazards or race conditions. These problems must be considered both at the hardware interface protocol level and the software level (compiler, operating system, and so forth).

Last, coprocessors that require explicit control from the host CPU often necessitate special resources, such as special pins or interconnection points with the CPU that would not normally be necessary. All of these impact the general performance of the host and must be considered to determine overall cost and effectiveness of adding a coprocessor to a system.

3.5. Chapter Summary

In this chapter we have reviewed some of the hardware and software issues related to the incorporation of coprocessors in computer systems. The primary consideration in selecting software applications that should use or be embodied in special purpose coprocessor hardware is based on an evaluation of the potential performance improvement produced. We have identified several factors that must be taken into consideration when making that evaluation and presented a model of the many variables affecting the outcome. The speedup is referred to as coprocessor effectiveness. Specific instances will call for refinement of the model to be useful. The efficiency of a coprocessor can be determined by the relative amount of time it spends performing useful work versus the time it is idle. Under-utilized hardware allows for the possibility of multiplexing in multiprocessor systems.

In the next two chapters, we will examine two applications that use coprocessor hardware dedicated to specific functions — floating-point arithmetic and dynamic programming optimization — and use aspects of the performance model developed here as a means of evaluating and comparing different implementations of each.

<This page is intentionally blank.>

4 Floating-point Arithmetic Coprocessors

4.1. Introduction and Overview

There are many problems in science and engineering that require the numeric range and precision afforded by floating-point arithmetic. Large mainframe computers are often designed expressly for numeric computation, or at least with the anticipation that it will be an important and integral part of the system. In many instances, the capability for floating-point arithmetic is likely to be standard equipment, rather than optional.

In Chapter 2, we saw how array processors and floating-point accelerators were added to computer systems to substantially improve performance. Such devices are still relatively expensive, being as much as 50% of the total cost of the host CPU. For some problems, the speed or capacity requirements simply can not be met by general-purpose CPU's, even with floating-point enhancements. In such cases, a super computer is the only answer. Nevertheless, there are a large number of floating-point intensive applications being run on microprocessor-based workstations. These include many aspects of computer aided design, circuit simulation, image synthesis and rendering, and signal processing. Such applications can be intolerably slow without good floating-point performance in the system. And yet one of the driving forces for the widespread use of workstations is the low relative cost. To meet both needs of performance and economy, a popular coprocessor — the floating-point unit (FPU) — is available on most microprocessor-based workstation systems.

This chapter considers VLSI floating-point arithmetic coprocessors and examines the interface styles of past and current implementations. We are interested in determining how effectively the floating-point coprocessor is exploited in each of the architectures and in contrasting that with the SPUR system.

Section 4.2 describes operational characteristics of the Intel i80286/i80287 and Motorola MC68020/MC68881 products listed in Table 2-3, and of the UC Berkeley SPUR FPU. The two commercial coprocessors are chosen because they represent two contrasting styles of interface, are the most

widely used, and both claim to allow concurrent execution of the CPU and floating-point coprocessor. They are also available in laboratory settings for the measurements needed to verify our analyses and results. Section 4.3 briefly outlines the approach we use in comparing and determining performance of the three systems and Section 4.4 considers several metrics for the evaluation. Section 4.5, by way of architecture models and run-time traces, isolates the effects of the details of each implementation from the architecture to allow a more meaningful comparison. Lastly, Section 4.6 summarizes this chapter and briefly considers interface requirements for future high-performance workstation systems equipped with floating-point coprocessors.

4.2. Intel, Motorola, and SPUR Floating-point Coprocessor Parameters

In this section we give a brief overview of the functionality of each of the systems. Specific details are provided in Section 4.4.3 relating to the analysis of our experimental results. The floating-point coprocessors described here are all capable of handling single-, double-, and extended-precision operands.

4.2.1. The Intel FPU

The Intel Numeric Data Processor (NDP) i8087 was the first commercially available device to implement a version of the ANSI/IEEE Standard P754-1985 for Binary Floating-point Arithmetic [Cody84, Inte81b, Inte85b, Nave80, Palm80] and Intel is likely to have coined the term *coprocessor* for this device. The newer Numeric Processor Extension (NPX) — i80287 or i80387 — are mostly object code compatible with the i8087. The inner cores of the i8087 and i80287 chips are identical, but the i80387 has been reengineered, uses a different interface protocol, is faster for many operations, and produces slightly different results in some cases. The chips provide add, subtract, multiply, divide, square root, and several transcendental functions (sine, cosine, tangent, $\log_2$, and others).

The i8087 is a synchronous instruction tracker to the i8086 [Inte85a], while the i80287 and i80387 appear as special I/O devices in a reserved address space, controlled by programmed I/O instructions [Inte87]. Figure 4-1 is a simplified block diagram of the Intel i80286/i80287 configuration.* In the first generation i8086/i8087 system, when a non-memory reference arithmetic instruction is encountered in the instruction stream (ESC opcode), both the i8086 and i8087 decode it, but the i8087 begins execution of the instruction immediately, while the i8086 continues to the next instruction. If the floating-point instruction includes a memory reference, the i8086 calculates the effective address for the first word of the operand, performs a dummy read, and transfers control to the i8087, which then completes the transfer with multiple memory bus cycles for each 16-bits of operand.

The second-generation math coprocessor from Intel, the i80287 is not an instruction tracker. Instead, the host i80286 decodes the ESC instruction and explicitly initiates an arithmetic operation in the i80287. No parallel decoding of instructions is possible since the i80286 maintains an on-chip instruction prefetch buffer and three-deep instruction queue. All diadic operations are between an operand in memory or an arbitrary register in the NPX register stack and the top element in the stack. All data transfers from memory are requested by the i80287 and satisfied by the i80286 CPU acting like a direct memory access channel to the i80287. This allows the memory management and protection mechanisms to be handled in a uniform manner in the i80286. Since the i80286 can proceed with integer instructions after a floating-point instruction has been dispatched, the architecture requires that the segment address and offset address of both the instruction and the operands for each floating-point instruction must be transferred to the i80287 before execution can begin. This is needed in order to to service any floating-point exceptions that occur, and always necessitates several bus cycles over the 16-bit data bus to the i80287.

*The Intel i80387 floating-point coprocessor became available for limited use following the completion of the work reported here. During our study, there was no opportunity to use one directly, and consequently, the results we present here are based on the systems available to us at the time.

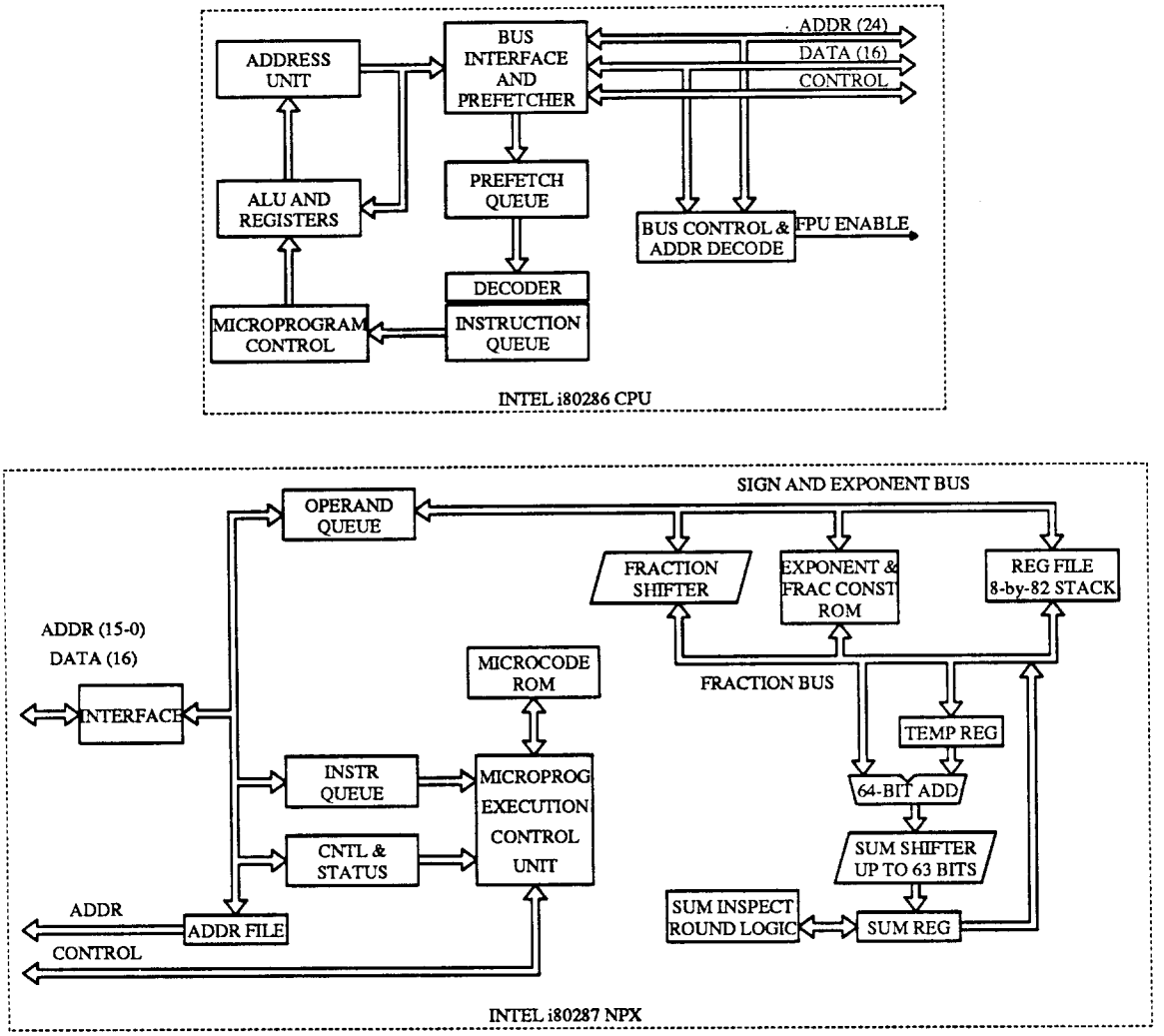


Figure 4-1. Block Diagram of Intel CPU-FPU-Memory System.

This is a simplified representation of the topology for the Intel system with numeric processor extension (NPX) floating-point coprocessor. The CPU acts as a DMA controller to pass operands between memory and the FPU.

To summarize, the i8087 and i80287 differ in both instruction fetching and data load/store mechanisms, although the arithmetic execution unit is identical for both. I/O and all communication between the i80286 and i80287 happen through special dedicated ports in the I/O address space. Intel compilers typically insert a WAIT instruction before every NPX coprocessor instruction to assure proper synchronization. However, every NPX instruction initiates a WAIT sequence in the microcode, suggesting that some inserted WAIT instructions are redundant. Nevertheless, WAIT's are necessary before any memory accesses by the CPU, and can not be eliminated. More details about the interface and function of the NPX can be found in [Inte85a].

4.2.2. The Motorola FPU

In March 1985, Motorola offered samples of the MC68881 floating-point coprocessor (FPCP) [Elec85] and went into production during the following summer.* The FPCP is designed to exploit the

*The second-generation Motorola MC68882 floating-point coprocessor became available following the completion of the work reported here. During our study, there was no opportunity to use one directly, and consequently, the results we present here are based on the systems available to us at the time.

coprocessor interface of the 32-bit CPU, which uses special control signals to implement the hardware interface protocols. The MC68881 implements floating-point add, subtract, multiply, divide, exponentiation, various logarithms, square root, and several transcendental functions (sine, cosine, tangent, hyperbolic functions, and others) [Cass84]. Figure 4-2 is a simplified block diagram of a Motorola CPU-FPU configuration.

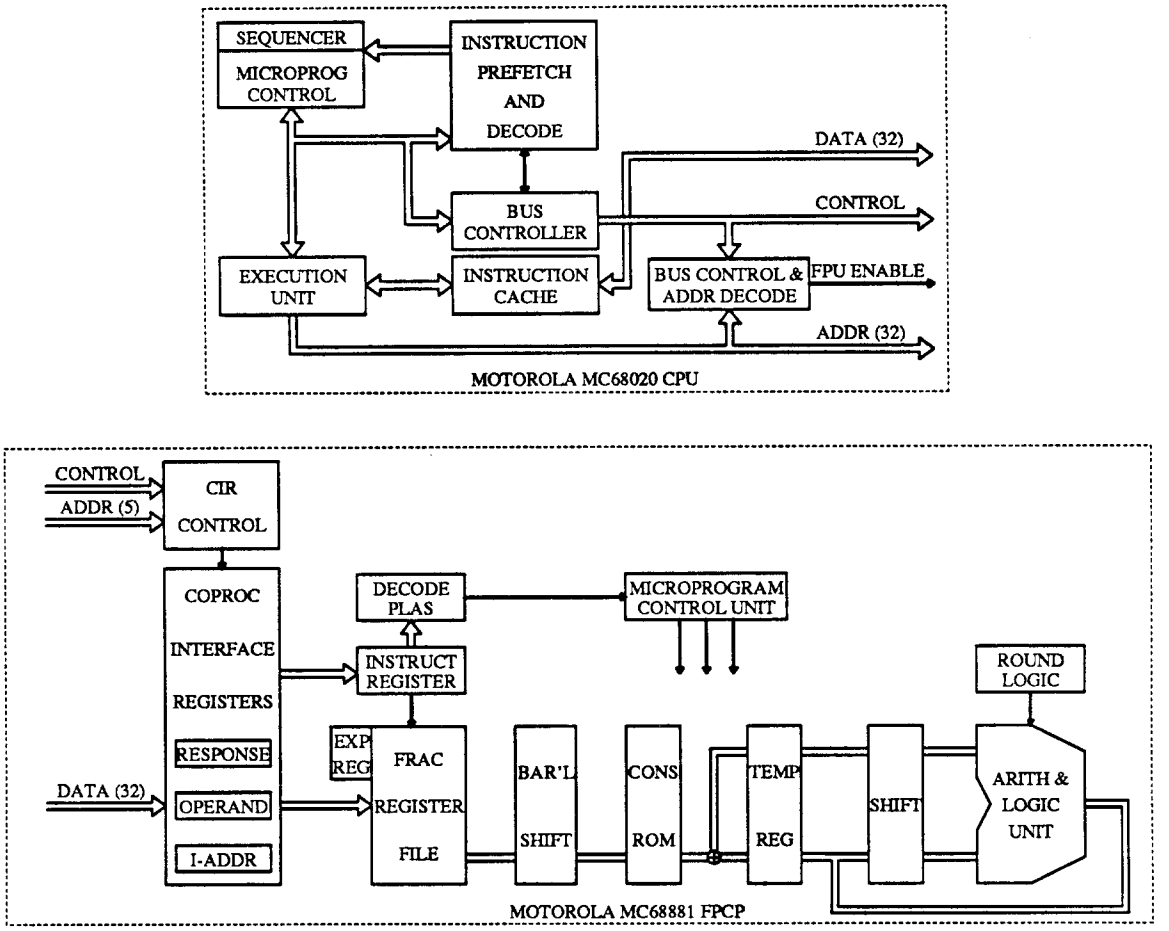


Figure 4-2. Block Diagram of Motorola CPU-FPU-Memory System.

This figure illustrates a simplified representation of the Motorola system with a floating-point coprocessor. All data transfers between memory and the coprocessor pass through special CPU registers requiring several bus cycles.

All communication between the CPU and FPCP uses standard bus cycles. Any other manufacturer's CPU can serve as a host processor, although without taking advantage of the coprocessor interface implemented in hardware. The MC68881 operates asynchronously to the main processor and does not track instructions. (Instructions are cached on the MC68020 CPU, making instruction tracking impossible since the instruction bus is not available at the CPU pins.) The FPCP is viewed as a special I/O device much like a memory-mapped peripheral. Logically, it appears as a set of locations in the memory map of the CPU. Future Motorola products are planned that will allow coprocessors to gain control and become local bus master [Cass84]. However, the FPCP requires all data transfers between memory and the coprocessor to pass through special 32-bit CPU registers. The FPCP and CPU operate in a virtual-memory, virtual-machine environment, with the host providing effective address calculation. The MC68881 supports all addressing modes of the MC68020. All exception handling is done by the main processor, with the coprocessor providing a floating-point exception vector.

If the FPCP requests and stores the CPU program counter for the current floating-point instruction (needed for possible exception processing), it can operate concurrently with CPU. To resynchronize

during concurrent operation, the CPU checks to see if the coprocessor is busy at the beginning of every coprocessor instruction, and suspends its operation until the FPCP is no longer busy, if needed. More details about the coprocessor interface and function of the Motorola floating-point unit can be found in [Moto85].

4.2.3. The SPUR FPU

The SPUR FPU is one of three custom VLSI chips in each node of a SPUR multiprocessor workstation. It performs add, subtract, multiply and divide for single, double, and extended precision operands. Other operations (sine, cosine, log, and so forth) are runtime library routines. Operations are done in extended format with conversions to other formats made implicitly on loads or explicitly before stores to memory.

The FPU is a load/store architecture with all operations between registers (two source and one destination specifiers). The address of floating-point instructions is saved in the FPPC register in the CPU for exception processing when needed. The SPUR architecture has no memory reference floating-point instructions. Thus, there is no need to save operand addresses for exception processing. Double-precision operands are transferred between the FPU and cache memory in a single bus cycle. The FPU monitors the 32-bit CPU instruction bus. Both the CPU and FPU decode instructions issued from the CPU instruction buffer simultaneously, so the FPU requires no explicit control from the CPU to begin operation.

The FPU is designed to operate concurrently with the CPU. Operand loads, stores, and CPU integer instructions can proceed while the FPU is busy. The CPU and FPU are completely synchronous; a busy/done signal provides status of the FPU execution unit for synchronization. Explicit synchronization can be programmed with the SYNCH instruction. More details about the SPUR floating-point coprocessor interface are included in [Hans86], summarized in Appendix A. Bose describes the microarchitecture and implementation of the FPU [Bose88b] and Lee outlines the implementation considerations of the IEEE arithmetic standard [Lee86]. Figure 4-3 shows a block diagram of a SPUR processor node with an FPU coprocessor.

4.2.4. FPU Summary

In summary, some of the characteristics and instruction execution times for the floating-point coprocessors discussed in this section are included in Table 4-1. A more complete list of the timings is found in Appendix B.

There are many differences between the three floating-point coprocessor implementations. The arithmetic execution rates are particularly interesting. Table 4-2 compares each of the implementations to the SPUR execution rate, based on the assumed clock rates shown in Table 4-1. From Table 4-2, we see that the speed ratios vary from 37-to-1 for floating-point add, 19-to-1 for floating-point multiply, to 11-to-1 for floating-point divide. The difference in performance stems from the complexity of operations provided and implementation strategy. As shown in Figures 4-1 and 4-2, the commercial coprocessors use microcoded control units and multiple-cycle iterative algorithms to implement the floating-point operations. SPUR uses direct, hardwired control. Also, the floating-point unit data path is reused for both fraction and exponent calculations in the commercial systems, whereas SPUR provides a separate exponent data path. Where the commercial implementations have used silicon to implement transcendental and other sophisticated functions in microcode, the SPUR FPU has generally used the technology to speed up fundamental operations. The tradeoffs involve a comparison of the frequency of certain operations in typical codes with the amount of time consumed to implement those operations.

As shown in the bottom half of Table 4-2, whether the operation is between a memory operand and a register, or simply between registers, the commercial FPU's are essentially identical. Floating-point add is roughly 1.1 to 1.4 times faster than floating-point multiply, and floating-point multiply is from 1.5 to 2.5 times faster than floating-point divide. For SPUR, the ratios are slightly different. Since there are only register-register operations, all execution delays are deterministic. Floating-point add is 2.5 times faster than floating-point multiply and floating-point multiply is 2.5 times faster than

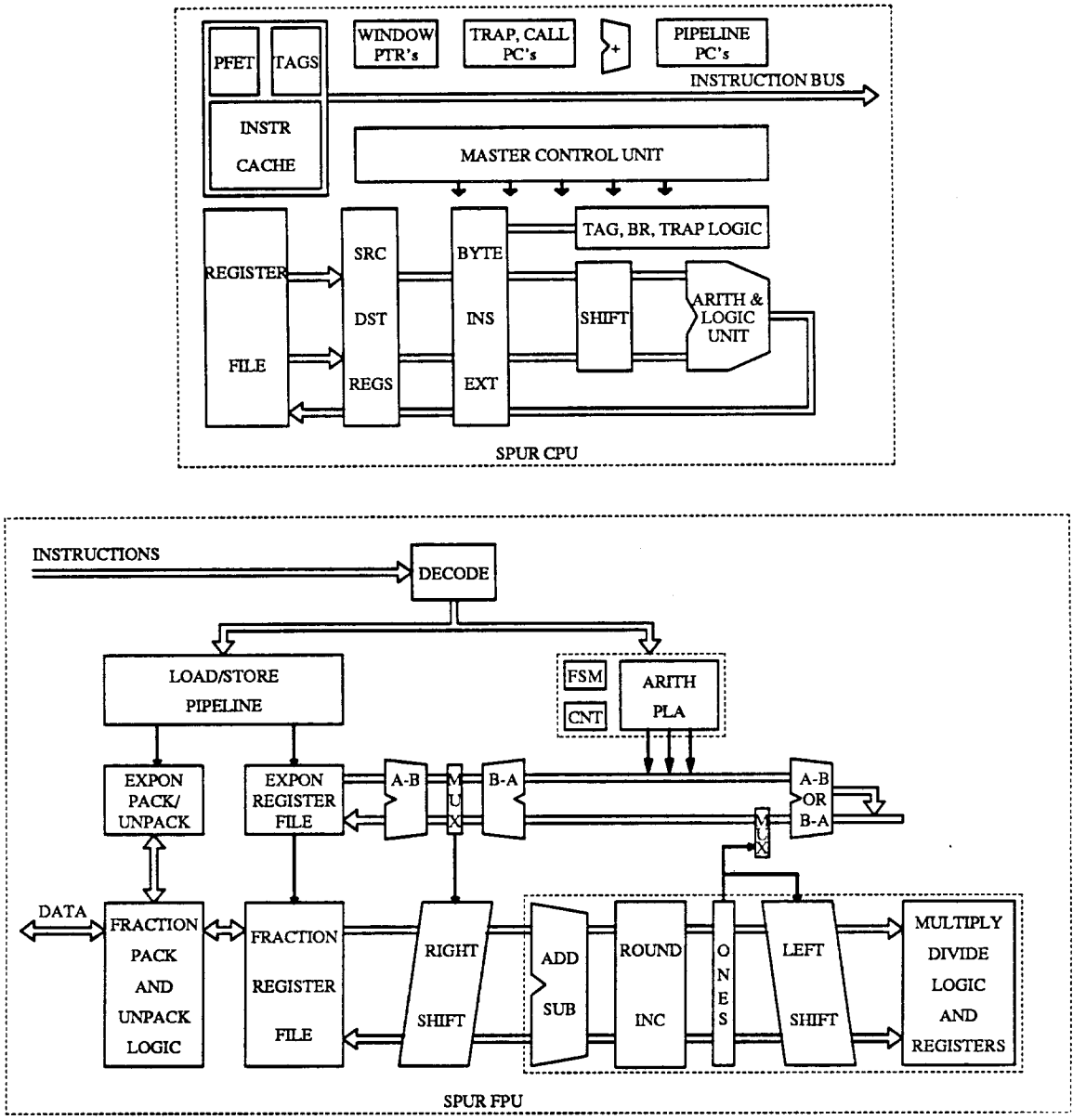


Figure 4-3. SPUR Workstation Processor Node.

This figure shows the major blocks of a SPUR CPU and FPU. The FPU coprocessor pipeline scheme is similar to that used by the SPUR CPU (see [Chen85]) and allows overlap between operand LD/ST, instruction fetch, execute cycles, and register result writes. The CPU maintains an on-chip FPU program counter to enable exception handling.

floating-point divide.

It is generally believed that if the product of execution time and operation frequency for basic operations is about equal, algorithm designers will not be tempted to use contorted code to either take advantage of a fast operation or avoid a particularly slow one. In the next section, we analyze some of the characteristics of floating-point programs to determine the frequency of operation occurrence to help us select some representative codes for subsequent analyses.

Table 4-1. Intel, Motorola, and SPUR FPU Parameters

Floating-point Unit	Intel i80287				Motorola MC68881				SPUR FPU	
Clock Frequency	16 MHz				16.7 MHz				6.7 MHz	
Execution Time	Cycles		Microsec		Cycles		Microsec		Cycles	Microsec
	R-R	M-R	R-R	M-R	R-R	M-R	R-R	M-R	R-R	R-R
FADD	140-200	190-250	8.8-12.5	11.9-15.6	47	77	2.8	4.6	3	0.42
FMUL	180-290	224-336	11.3-18.1	14-21	67	97	4.0	5.8	8	1.1
FDIV	386-406	440-460	24.1-25.4	27.5-28.8	99	109	5.9	7.4	19	2.7
FLD/FMOV	34-44	80-120	2.1-2.8	5-7.5	33	58	2.0	3.5	1+	.14
FST/FMOV	30-44	192-208	1.9-2.8	12-13	33	86	2.0	5.2	2+	.28
FCVT	na	incl	na	incl	na	incl	na	incl	3	.42
Enable FPU	Address; 12 Cntl lines				5 Addr, 7 Cntl lines				1 Cntl line	
Clocking	Asynchronous				Asynchronous				Synchronous	
Bus Protocol	Master				Slave				Concurrent	
Data Transfer	DMA				Prog I/O				Prog I/O	
Path Width to Mem	16-bit				32-bit				64-bit	
No. Operand Specs	1 + TOS				2				3	
No. GP Registers	8				8				16*	
No. Address Modes	All (8)				All (18)				All (2) LD/ST	
No. H/W Excepts	6				8				7	
Exception Detected	Next FPU OP				Next FPU OP				Immediately	

This table lists typical execution times for double precision (64-bit) register-register and memory-register operations in cycles and microseconds. All FPU's support extended precision operands internally, with conversions from/to other formats either explicitly or implicitly on LD/ST operations. The Intel and Motorola devices maintain eight user-accessible registers, while the SPUR FPU provides 16 (one register is defined as zero, 14 are 87-bit general purpose operand registers, and one is the floating-point status register). [Bose88b, Inte85b, Pren87].

Table 4-2. Comparison of Intel, Motorola, and SPUR FPU Execution Rates

Floating-point Unit	Intel i80287		Motorola MC68881		SPUR FPU
Clock Frequency	16 MHz		16.7 MHz		6.7 MHz
Operation Time vs SPUR	R-R	M-R	R-R	M-R	R-R
FADD	21.0 - 29.8	28.3 - 37.1	6.7	11.0	1.0 (0.42 μsec)
FMUL	10.3 - 16.5	12.7 - 19.1	3.6	5.3	1.0 (1.1 μsec)
FDIV	8.9 - 9.4	10.2 - 10.7	2.2	2.7	1.0 (2.7 μsec)
Operation Time vs FMUL					
FADD	0.7 - 0.8	0.7 - 0.9	.7	.8	.4
FMUL	1.0 - 1.0	1.0 - 1.0	1.0	1.0	1.0
FDIV	1.4 - 2.1	1.9 - 2.3	1.5	1.3	2.5

4.3. Determining Floating-point Coprocessor Performance

Floating-point coprocessors clearly enhance the performance potential of a computer system. Often it is possible to achieve one to two orders of magnitude improvement in performance over software by using one [Inte81a, Patt84, Sipp82]. However, unless careful attention is given to other factors at the level of overall system design, much of the advantage of having the coprocessor can be lost to overhead required in initializing and using the device, as was the case for one instance of using an array processor described in Chapter 2. In this section we focus on interface issues, and illustrate some of the inherent weaknesses of the coprocessor architectures described above and propose solutions to those problems. We start by briefly considering the characteristics of floating-point arithmetic programs. Then we present a simple set of benchmarks based on some common floating-point routines, measure the performance of the three systems, evaluate the factors determining the performance, and

compare the effectiveness of each implementation through the use of architecture models and simulation.

4.3.1. Characteristics of Floating-point Computation

Dongarra *et. al.* describe the process of measuring the power of computer system as "an art at best" [Dong87]. They suggest that effective benchmarking should include three elements: accurate characterization of the workload, initial tests using simple programs, and further tests with programs that approximate ever more closely the jobs that are part of the workday. To briefly characterize the nature of floating-point workloads, we summarize an analysis of the Livermore Loops by [Hans85], a report by [Leun86] analyzing the SPICE circuit simulator and Lattice filter simulator at Berkeley, work by [Bose88b] and the open literature [Curn76, Gibs70, Knut71]. Table 4-3 lists the frequency of various floating-point operations normalized to the occurrence of floating-point multiply.

Table 4-3. Operation Frequency for Various Benchmarks and Programs

Function	Program or Benchmark							
	Livermore Loops	Lattice Filter	SPICE + Trans	SPICE Decomp	Knuth FORTRAN	Gibson Mix	Whet-stone	Ave
Add/Sub/Cmp	1.86	0.75	1.68	1.47	2.3	1.8	1.8	1.7
Multiply	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.0
Divide	0.10	0.083	0.34	0.38	0.38	0.39	0.50	.4
Square root	0.03	-	0.04	-	-	-	-	nil
Transcendentals*	-	-	0.034	-	-	-	-	nil

This table summarizes the frequency of operation of floating-point arithmetic for some well known benchmarks and programs. The column head by "SPICE Decomp" is the result of decomposing the transcendental functions into basic operations. All entries are normalized by the frequency of floating-point multiply. *Transcendentals include sine, cosine, arctangent, exponentiation, $\log_2$, $\log_{10}$, and so forth.

From Table 4-3, we note that add/sub/compare operations occur about 1.7 times as often as multiply, and that multiply is usually between two and three times as frequent as divide. Although it would appear that transcendentals account for a vanishingly small fraction of all operations, it was determined in [Leun86] that when decomposed into fundamental operations (i.e., add/sub, multiply, and divide), the number of occurrences for the fundamental add/sub/compare operations increased by a factor of 1.87, multiply increased by 2.14 times, and divide increased by 2.28 times for the SPICE program.

By combining the information from Table 4-2 and Table 4-3, we produce Table 4-4, which shows frequency-delay product for the three CPU-FPU pairs and the basic arithmetic operations.

Table 4-4. Frequency-delay Product for Intel, Motorola, and SPUR

Operation	Frequency-delay Product (double-precision operands)				
	i80287		MC68881		SPUR
	R-R	M-R	R-R	M-R	R-R
FADD	1.2 - 1.4	1.2 - 1.5	1.2	1.4	.7
FMUL	1.0 - 1.0	1.0 - 1.0	1.0	1.0	1.0
FDIV	.56 - .84	.76 - .92	.60	.42	1.0

From this we observe that floating-point add is 20% to 50% slower than the optimum frequency-delay product for the commercial systems. On the other hand, floating-point divide is between 1.1 and 2.4 times faster than it needs to be relative to floating-point multiply. This suggests that if resources

expended in making floating-point divide execute efficiently could have been used to increase the speed of floating-point add, overall the system would be more cost effective, yielding a better average performance.

For SPUR, the frequency-delay products are in balance, with floating-point add being 30% faster than the predicted optimum. Again, if resources can be traded, a slightly slower floating-point add with correspondingly faster floating-point multiply and floating-point divide operations would make SPUR more closely fit the frequency-delay statistics. In the next section, we consider some typical floating-point routines to use as benchmarks in our comparative analysis.

4.3.2. Microbenchmarks — The Floating-point Arithmetic Claim

To identify some simple programs that could serve for initial tests, we conferred with Professor W. Kahan of the University of California, Berkeley, an authority on floating-point arithmetic and one of the originators of the IEEE floating-point standard. He suggested that floating-point applications can largely be represented by simple equations inside loops [Kaha85]. These represent the kernels from Gaussian elimination or LU decomposition used in matrix inversion, linear algebra, and so on; scalar or dot product and partial fraction expansion; polynomial evaluation using Horner's Rule or continued fraction equations. These are identified as GE, DP, and PE* and can be expressed in simple forms where X, Y, C, and D are vectors of floating-point numbers, K is a floating-point constant, P represents intermediate partial results from a floating-point computation, and n ranges from tens to thousands.

Gaussian Elimination (GE)	**for** i = 1 to n **do** Y[i] = Y[i] + (K * X[i])	(4-1)
Dot Product (DP1)	**for** i = 1 to n **do** P = P + (X[i] * Y[i])	(4-2a)
Partial Fraction (DP2)	**for** i = 1 to n **do** P = P + (X[i] / (Y[i] - K))	(4-2b)
Polynomial Evaluation (PE1)	**for** i = 1 to n **do** P = (P * K) + C[i]	(4-3a)
Polynomial Evaluation (PE2)	**for** i = 1 to n **do** P = P * (K - X[i]) + C[i]	(4-3b)
Continued Fraction (PE3)	**for** i = 1 to n **do** P = (K / P) + C[i]	(4-3c)
Continued Fraction (PE4)	**for** i = 1 to n **do** P = (D[i] / P) + C[i] + K	(4-3d)

Kahan's assumptions correlate with [Robi76] and [Koba84] who show that statically between 60% and 70% of all floating-point arithmetic involving arrays of operands is computed inside loop nests. Further, dynamic measurements indicate that virtually all computation is performed in loops

*The LINPACK Users Guide [Dong79] contains extensive listings of programs for the solution of linear systems and related problems. DAXPY is one of the Basic Linear Algebra Subprograms (BLAS [Dong79]) used by LINPACK routines, and is essentially the same as GE above. In [Dong88], the amount of time needed to execute one iteration of DAXPY is referred to as the *Unit*. The BLAS also contain DDOT, which is used in the forward- and back-substitution steps of LU decomposition, and is essentially the same as DP above. Knuth [Knut75] gives credit to Sir Isaac Newton for the first formulation of PE.

[Knut71] and the majority of all loops in FORTRAN programs are small, being two statements or less [Knut71, McMa86, Robi76]. Indeed, Knuth concluded that, "A small number of basic patterns account for most of the programming constructions in use."

To see how these compare with the analyzes reported in Table 4-3, Table 4-5 summarizes the frequency of operations for the set of kernel benchmarks in (4-1) through (4-3). Here, we have grouped similar algorithms into three categories: GE, DP, and PE.

Table 4-5. Kernel Benchmark Floating-point Characteristics

Loop Name	Add/Sub	Multiply	Divide	Read Mem	Write Mem	FP Ops per Mem Ref
GE	1	1	0	2	1	0.67
DP	1.5	.5	.5	2	0	1.25
PE	1.5	.25	.75	1.5	0	1.67
Average	1.33	0.58	0.42	1.67	.33	1.12
Relative to Multiply	2.3	1.0	0.72	-	-	-

This table summarizes the frequency of operation of floating-point arithmetic for some small kernel benchmarks. The composite characteristics for the three groups are shown, with the averages correlating with the results in Table 4-3.

From this, a simple yet representative suite of benchmark programs would be (4-1), (4-2a), and (4-3c), where the ratios of add/sub/cmp to multiply to divide is 4 to 2 to 1. However, to illustrate a wider variation in performance, and highlight some of the architecture features studied in this dissertations, we replaced (4-3c) with (4-3a). It is a simple matter to extrapolate our final results to the other cases. The following section describes the performance tests of the Intel, Motorola, and SPUR CPU-FPU pairs running simple codes that implement these microbenchmarks.

4.3.3. The Floating-point Experiment

First, small programs were written in a high-level language for each of GE, DP, and PE. These programs were then translated to assembly language code with the best compilers available on real machines, always employing the optimization phase if available. Each assembly language listing was examined and modified to make maximum use of registers for all architectures. The code was then assembled and run to guarantee correctness. This code is referred to as the *FORT* version.

Second, each of the programs was rewritten using pointer variables, then translated and hand optimized for register use as before. Using pointers often reduces the amount of redundant address arithmetic found in the FORT version, and is typical of C programs. This code is referred to as the *CPTR* version.

Third, each program was hand optimized in assembly language to eliminate redundant jumps, no-ops, and other unnecessary calculations found in previous versions. Each program was tuned to take advantage of the architecture of the machine it was running on, allowing for instruction prefetch, overlap, and other forms of parallelism whenever possible. Simple code-motion optimizations and loop unrolling were used to increase performance in some cases. This version is called *ASSM*. (The high-level language and some of the assembly versions of the code are listed in Appendix B for the interested reader.)

The programs (27 in all) were run on native architectures or simulated for timing. Simulation models account for all architecture features and reflect accurately the timing associated with running programs. The value for n can vary from 10 to several thousand. A key observation is that the loops achieve a first-order steady state behavior (ignoring page faults) that makes it possible to examine just a few iterations and accurately extrapolate to larger n. This is based on a worst-case cache behavior (i.e., a linear walk through memory for data references) and represents a lower bound on performance.

The running time for each of the programs and systems is given in Table 4-6.* To facilitate comparison, the results are normalized to the slowest execution time for all programs and versions. Since these numbers represent execution times, those less than 1.0 indicate better performance. Figure 4-4 is a simple bar chart of the data shown in Table 4-6. The Y-axis indicates the fraction of the Intel CPU-FPU performance for dot product (FORT version) that each of the CPU-FPU pairs achieves.

Table 4-6. Absolute Execution Time Normalized by Intel FORT Version of DP

Program	Intel i80287			Motorola MC68881			SPUR FPU		
Version	GE	DP	PE	GE	DP	PE	GE	DP	PE
FORT	0.966	1.000	0.937	0.311	0.308	0.161	0.034	0.031	0.020
CPTR	0.951	0.994	0.967	0.289	0.264	0.157	0.031	0.028	0.020
ASSM	0.855	0.684	0.650	0.279	0.170	0.151	0.029	0.020	0.020

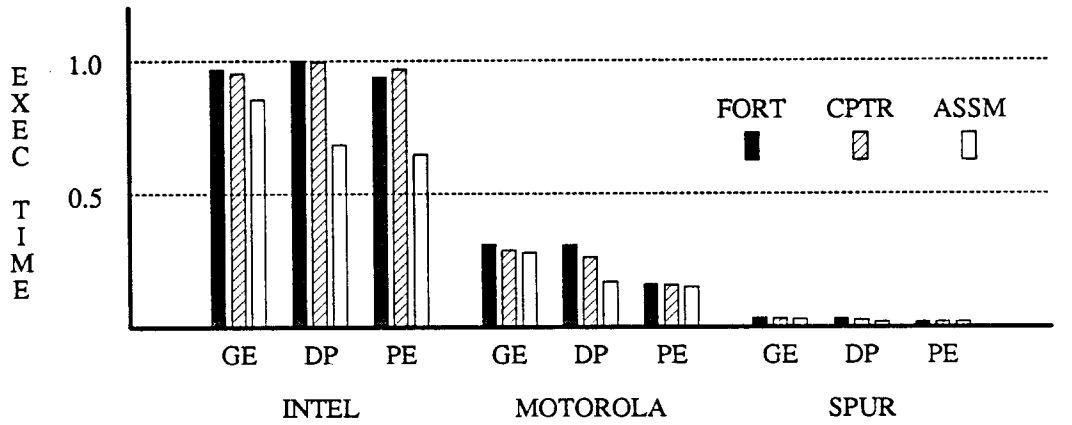


Figure 4-4. Absolute Performance Comparison of Three CPU-FPU Pairs.

This table and figure show the absolute execution time for the three microbenchmarks: Gaussian elimination (GE), dot product (DP), and polynomial evaluation (PE), for three workstation computer systems using VLSI floating-point coprocessors. Execution times are divided by the Intel FORT execution time for DP (74.6 microseconds per loop iteration with a 16 MHz system clock). Typically, the CPTR version was slightly better than the FORT version. For Intel PE, however, the code generated by the C compiler was slightly less efficient, resulting in a small performance degradation. Numbers less than 1.0 indicate relatively faster execution.

It is interesting to note the differences between the FORT or CPTR versions and the ASSM version of each of the programs for each of the system. Intel shows an 11% to 13% reduction in execution time for assembly code compared to compiled code. Motorola's assembly version execution time is 11% to 45% less for GE and DP. SPUR hand-assembled versions are between 0% and 36% less execution time compared to compiler-generated code. The biggest gain comes from efficiencies achieved by unrolling loops.

The hand-assembled version of DP for Motorola takes advantage of two features of the architecture to achieve the improvement. First, rather than holding the partial sum P (see Equation 4-2) in memory, as the compiler-generated code specifies, an on-chip register is used to accumulate it. Second, the Motorola architecture allows general register specifications of floating-point operations, enabling the floating-point register to be used as an accumulator. The Intel architecture also profits from the

*As shown in Table 4-1, the clock rate for both the Intel i80286/i80287 and Motorola MC68020/MC68881 was assumed to be 16.7 Mhz, and the SPUR clock rate was assumed to be 6.7 Mhz. Recent versions of some of these run at clock rates 30% to 40% faster. Nevertheless, the ratios shown in Table 4-6 would remain roughly the same.

stack-like register file, leaving the partial sum in the top-of-stack register and pushing the array variables to be multiplied before the accumulating addition. More will be said about this in Section 4.3.

In some cases, there is very little difference between the compiled and assembled versions. This might suggest that either the compilers are very good, the hand-optimized code is not very good, or that the simplicity of the program leaves little room for optimization. In fact, several things contribute to this difference. First, the so-called compiled versions have been modified to the "best common denominator" to remove obvious inefficiencies, such as the elimination of redundant or repeated instruction sequences. This tends to make the compiled versions more efficient that would normally be the case for some of the systems.

Second, the amount of time spent with floating-point instructions far and away dominates the time spent on other instructions. Eliminating integer instructions has little overall effect in some cases. Also, it is difficult to influence the performance by substituting alternative floating-point codes. For example, the only possible modification to a floating-point multiply instruction with a memory reference is to exchange it for two instructions: an explicit floating-point load followed by the register-register floating-point multiply. In practice, no improvement in execution time is seen as a result for either Intel or Motorola.

Third, as mentioned earlier, with the small number of instructions usually found in tight inner loops, the possibilities for improving performance through any optimizations are limited. For the rest of our analyses, we will refer to two versions of code — compiled (designated COMP) and hand-assembled (designated ASSM) — to simplify and yet preserve the ability to contrast the two. COMP represents a combination of FORT and CPTR results.

4.4. Analysis of Floating-point Coprocessor Performance

In the previous section, we found that there is a large difference in absolute performance between the three systems. There are some obvious reasons: 16-bit versus 32-bit versus 64-bit data paths, for example. In the following sections, we analyze each of the systems in some detail to determine what is provided by each of the architectures, how each has been implemented, and the interaction of the architectures with their implementations.

4.4.1. Relative Performance Metric

Figure 4-4 plots speed of all programs on an absolute scale, normalized to make the comparison more convenient. A more relevant comparison is a one-for-one match of code versions; i.e., compare the COMP version of GE on Intel to the COMP version on Motorola and SPUR. Likewise for the other programs and versions. Table 4-7 summarizes this comparison, which is illustrated in Figure 4-5. Since all measurements are reported relative to Intel, it's results should be shown as unity (1.0) for all versions of all programs. But, to visualize the change that is brought about by hand optimization, the Intel ASSM version is shown relative to its COMP version.

Table 4-7. Execution Time Relative to Intel for All Versions and Programs

Program	Intel i80287			Motorola MC68881			SPUR FPU		
	GE	DP	PE	GE	DP	PE	GE	DP	PE
COMP	1.000	1.000	1.000	0.322	0.308	0.172	0.035	0.031	0.021
ASSM	0.885*	0.684*	0.694*	0.289	0.170	0.161	0.030	0.029	0.021

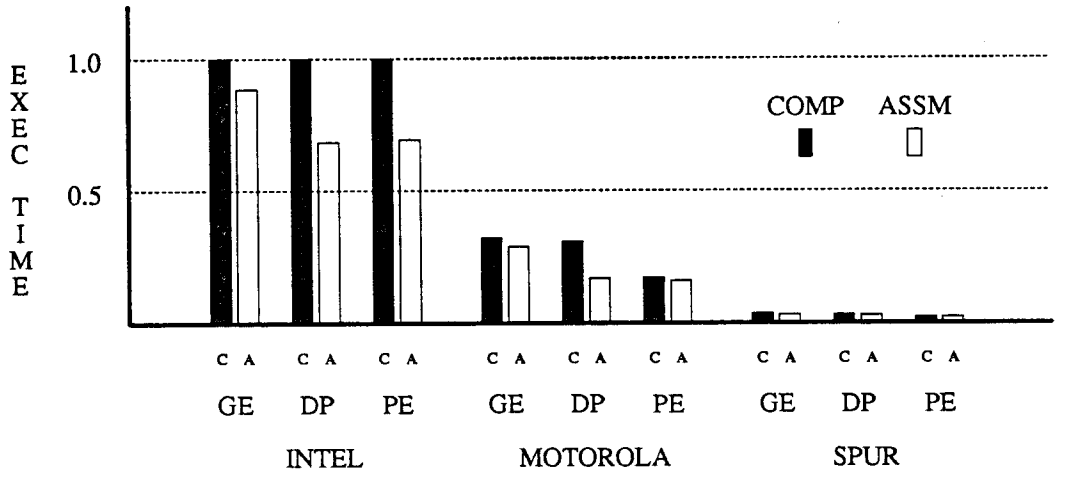


Figure 4-5. Relative Performance Comparison of Three CPU-FPU Pairs.

From the data in Table 4-7, this figure shows relative performance for the same three programs illustrated in Figure 4-4. COMP represents a combination of the FORT and CPTR results shown in Table 4-6. Here, the execution times for all versions of all programs are divided by the Intel execution times. For example, the COMP version of GE for all systems is divided by execution time for the Intel COMP version of GE. Similarly, the ratios are calculated for the ASSM versions of GE, as well as for all versions of all other programs. However, the ASSM versions of Intel are shown relative to the COMP version of Intel, whereas the ASSM versions of Motorola and SPUR are relative to Intel ASSM.

From the data summarized in Table 4-7, we note that the SPUR-FPU system is approximately seven to 10 times faster than the Motorola MC68020/MC68881 system, and between 30 and 50 times faster when compared to the Intel i80286/i80287 system.* These results are not surprising. The wide performance variation for the three systems can be attributed to a several factors. First and foremost, it is important to recognize that the Intel FPU is essentially a first-generation implementation, while the Motorola and SPUR FPU designs represent second generation architectures and implementations. As discussed earlier, the arithmetic execution unit of the i80287 is identical to the i8087, a late 1970's design. As a consequence, operation times are relatively slow in comparison to the other FPU's (see Appendix B).

Second, the physical characteristics of the system can have a considerable influence on the observed performance. For example, the bandwidth to memory determines the number of bus cycles necessary to transfer floating-point operands. Since a bus cycle is a multiple clock-tick event in most systems, this will result in reduced performance for any system with a relatively narrow bus and relatively wide data (e.g., double-precision floating-point).

Although comparing absolute execution time is interesting and informative, it is important to find ways to make meaningful comparisons that shed light on architecture issues, avoiding the coloring effects of the particular implementation. Also, there are limitations as to what can be concluded from comparison of small abstract models, and one must be careful when making generalizations based on such comparisons — particularly in our case where the comparison is based on three small programs.

We would like to find a means of identifying, quantifying, and if possible *normalizing* the effects of the various factors to be able to make a more meaningful comparison. We believe that an analysis of performance based on identifying those parts of the program that are directly related to floating-point execution versus all other parts provides insight into the effectiveness of a particular floating-point coprocessor architecture and its interface to the rest of the system. This is the topic of consideration in

*Section 4.4 discusses many of the architecture- and implementation-related reasons for the large difference in execution rate for the various floating-point units.

the following sections.

4.4.2. Floating-point vs. Non-Floating-point Instruction Metric

For floating-point arithmetic, performance measures are often based on *instruction times*, for example, how fast a double-precision multiply can be executed. But, real applications require interactions with memory and instructions other than the FPU instructions to implement algorithms. The performance numbers associated with floating-point execution given in data books may include the overhead associated with the operation and memory access, but obviously do not account for stalls due to data cache misses or the integer portion of the program associated with the floating-point operations. They also do not separate or identify the floating-point *operation time* and floating-point *operation overhead* relative to the entirety of the function being computed.

As a first step in our analysis of the comparative execution time data in Table 4-7, to help us understand why nearly two orders of magnitude difference in performance exist, we show the relative amounts of time spent in either floating-point or non floating-point instruction execution for the three systems for both the COMP and ASSM versions in Table 4-8 and Figure 4-6. For now, we ignore the influence of cache misses (i.e., assume a 0% data miss ratio).*

Table 4-8. Fraction of Execution Time Consumed by FP and Non-FP Instructions

Program Version	Param-eter	Intel			Motorola			SPUR		
		GE	DP	PE	GE	DP	PE	GE	DP	PE
COMP	FP Instrs	0.951	0.941	0.976	0.895	0.882	0.885	0.868	0.853	1.000
	Non-FP Instrs	0.049	0.059	0.024	0.105	0.118	0.115	0.132	0.147	0.000
ASSM	FP Instrs	0.983	0.969	0.978	0.937	0.963	0.928	0.938	1.000	1.000
	Non-FP Instrs	0.017	0.031	0.022	0.063	0.037	0.072	0.062	0.000	0.000

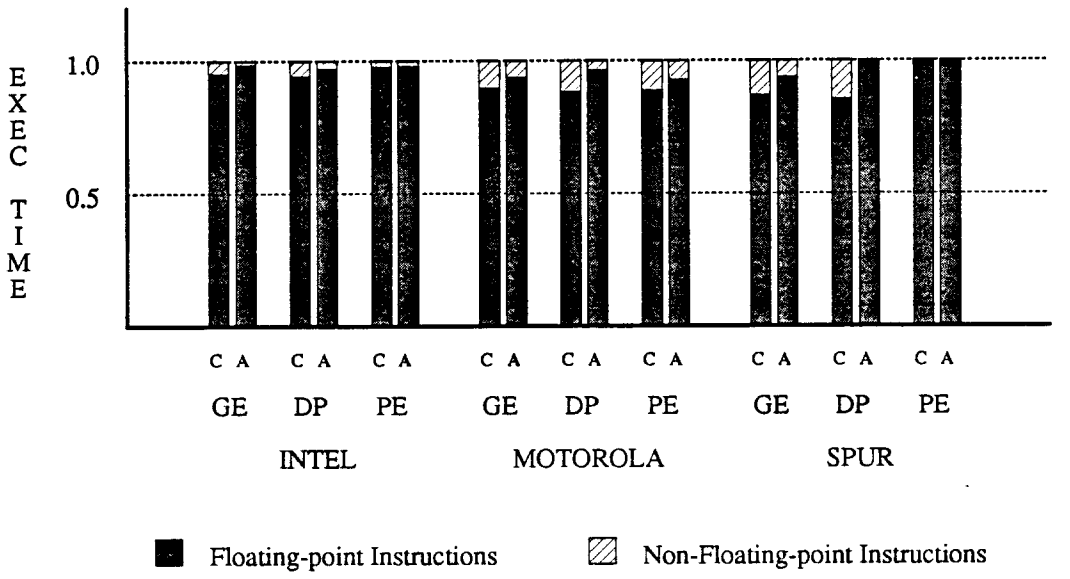


Figure 4-6. Relative Floating-point vs. Non-Floating-point Instruction Execution Time.

This figure shows the dynamic ratio of floating-point to non-floating-point instruction execution time. The left bar of each pair represents the COMP version, and the right bar represents the ASSM version of each program, as designated below each bar with "C" or "A".

*We will see in Section 4.5.3 that this is an unrealistic assumption. Modern systems with large virtual address spaces nearly always relay on a memory hierarchy, which often involve at least one level of caching. Also, Section 4.5 looks at implementation effects in general on system performance.

Figure 4-6 suggests that the time spent for floating-point instructions exceeds that spent on non-floating-point instructions by one to two orders of magnitude in some cases. A high ratio of floating-point to non-floating-point time would seem to be a desirable, since one would suppose that with the relatively large amount of time being spent on floating-point instructions, the system is performing well, keeping the hardware utilized effectively. However, this may be misleading. If the floating-point operations are relatively slow, the fraction of time spent on floating-point computation can be large, but overall the system performance might be comparatively poor. This is an undersirable effect. On the other hand, the amount of overlap between the CPU and FPU execution can reduce the *effective* non-floating-point execution time to a small amount, giving rise to a high floating-point to non-floating-point ratio. This is a useful and desirable effect. In order to understand and identify the factors that determine overall performance and account for *effective* as well as *ineffective* cycles, we will examine the execution of floating-point instructions in each of the architectures in the following sections.

4.4.3. Floating-point Instruction Components

Floating-point instructions, unlike most integer instructions, typically consume many clock-ticks during execution. Microcoded FPU's employ the typical time-space tradeoff by re-using portions of the data path and hardware to perform different portions of the operation, and in the process consume more cycles. Such an approach provides a means to "easily" implement more complicated functions, such as transcendentals, logs, square root, and others through microcoded algorithms. On the other hand, hard-wired FPU's achieve faster execution by dedicating more silicon to the fundamental operations and not providing direct implementation of the more complicated yet less frequent operations. Instead, software routines using the fundamental add, subtract, multiply and divide primitives are used. Although the microcoded design style provides hardware support for more arithmetic functions, the fundamental operations are typically slower in comparison to a four-function FPU. The overall effect on system performance is application dependent.

Besides the intrinsic operation specified by the instruction, several other events between the CPU and FPU must take place before an operation can begin. Each of the three systems has its own semantics for floating-point instructions. We will considered as an example a double-precision floating-point multiply instruction involving a memory reference operand and a floating-point coprocessor register.

In the Intel architecture, floating-point instructions operate on the top-of-stack (TOS) element of a stack-like register file. The operation is specified between the TOS and another floating-point register or memory operand. The *FMUL MEM_ADDR* instruction causes an operand in memory to be transferred to the FPU, where it is multiplied by the value in the TOS register, with the result left in the TOS. For Motorola, the *FMUL MEM_ADDR*, FP_{Rsrc} instruction is slightly more general. The register file is general purpose and any one of the eight FPU registers can be designated as an operand in a dyadic instruction with either another FPU register or a memory referenced operand. The result is stored in the source operand register. For SPUR, the operand is first transferred from memory to the FPU with the *LD_DBL* FP_{Rsrc1}, *MEM_ADDR* instruction. Then the floating-point multiply between it and any other register-based operand is completed with *FMUL* FP_{Rdest}, FP_{Rsrc1}, FP_{Rsrc2}. SPUR provides full generality in terms of which registers provide operands or accept results.

None of the architectures provides pipelined use of the floating-point execution unit. Thus, only one floating-point instruction is being performed at any time, and any new floating-point instructions can not begin until the previous one is finished. Also, Intel and Motorola allow full generality of addressing modes with floating-point instructions. Specifically, they provide for memory reference floating-point operations in the instruction set. These two facts combine to enforce a strict serialization of operand references and the operations on them, resulting in delaying any memory activity until the floating-point execution unit is no longer busy. We will consider the effects of this on overall performance in Section 4.4.4.

SPUR is a load/store architecture and references to floating-point data are decoupled from the operations on the data. The SPUR memory interface architecture allows floating-point operand loads and stores to proceed while the arithmetic execution unit is busy. More specifically, the SPUR FPU

LD_DBL and *ST_DBL* instructions are not included in the floating-point instruction set in the sense of the busy-test and synchronization. They are treated like any other integer instruction by the FPU and proceed in parallel with floating-point execution, even when the floating-point unit is busy. As a consequence, the operand accesses needed to support floating-point operations are completely decoupled from the instructions which operate on them. This concurrent execution model does not restrict the SPUR architecture to serialize its loads or stores, as is the case with the Intel and Motorola architectures.

During the process of transferring an operand from memory and performing the multiply, each hardware implementation imposes certain protocols regarding handshake signals, delays, busses, and registers used. Table 4-9 identifies these and quantifies their cost in terms of clock ticks for the *FMUL memory reference* instruction for each architecture. This list is a combination of all architectures, so each individual FPU will require only a subset of the items listed. The following paragraphs define the terms used in Table 4-9

IF(i), IF(fp) — instruction fetch

For many CPU architectures, by either prefetching and/or instruction-caching, the cycles normally associated with fetching an instruction can be "buried" during the execution phase of the previous instruction. Thus, the cycle cost can be "zero," assuming that the ALU execution time of the instruction accounts for it. The SPUR CPU microarchitecture uses a four-stage pipeline with an on-chip instruction buffer cache. Consequently, during any clock tick, four actions are completed. In this sense, an instruction fetch accounts for one-fourth clock tick, or at the end of the four clock ticks, a full fetch cycle has been accounted for.

WS(m) — memory wait state

Commercial computer systems must often accommodate different speed memories. As well, a particular VLSI CPU may be available at different clock rates. Accordingly, synchronization at the CPU-memory interface is necessary, with some form of handshake to determine when data are valid. In contrast, the SPUR system operates synchronously with its cache memory and requires no handshake synchronization cycles.

SYNC(fp) — synchronization busy/wait on FPU

Only one operation can be in progress at a time in the FPU's studied here. Each of the FPU's provides a signal or register value that can be interrogated by the CPU to determine status and prevent initiating another FPU instruction while the FPU is still busy.

IW(fp) — instruction/command write to FPU

In non instruction-tracker systems, the FPU command must be explicitly written to the FPU.

WS(fp) — coprocessor wait state

In non instruction-tracking systems, the process of writing the FPU command, instruction or data addresses, or operands to the FPU may require some local-bus arbitration and/or wait-states to synchronize the transfer. Also, commercial architectures allow the CPU and FPU to operate at different clock rates. This necessitates some form of synchronization for proper data, command, and status transfers.

Table 4-9. Floating-point Unit Hardware Protocol Events for Double-precision FMUL

Event Symbol	Event Description and/or Comment	Clock Ticks Consumed		
		Intel	Motorola	SPUR
IF(i)	instruction fetch - integer operation	0 - 4	0 - 2	1*
WS(m)	memory wait state	0 - 8	0 - 3	
IF(fp)	floating - point command fetch	0 - 4	0 - 2	1*
WS(m)	memory wait state	0 - 8	0 - 3	
SYNC(fp)	synchronization busy/wait on FPU	12 - 30	3 - 5	
IW(fp)	instruction/command write to FPU	4 - 8	3 - 5	
WS(fp)	coprocessor wait state	0 - 4	0 - 3	
RR(fp)	interrogate FPU for status	18 - 24	3 - 5	
IAW1(fp)	instruction address (offset) write to FPU	4 - 8	5	
WS(fp)	coprocessor wait state	0 - 4	0 - 3	
IAW2(fp)	instruction address (segment) write to FPU	4 - 8		
WS(fp)	coprocessor wait state	0 - 4		
OR1(fp)	operand 1 read (LSB)	4	4 - 6	(1)*
WS(m)	memory wait state	0 - 8	0 - 3	
OW1(fp)	operand 1 write (LSB) to FPU part 1	4	4 - 6	(1)*
WS(fp)	coprocessor wait state	0 - 4		
OR2(fp)	operand 2 read	4	4 - 6	
WS(m)	memory wait state	0 - 8	0 - 3	
OW2(fp)	operand 2 write to FPU part 2	4	4 - 6	
WS(fp)	coprocessor wait state	0 - 4		
OR3(fp)	operand 3 read	4		
WS(m)	memory wait state	0 - 8		
OW3(fp)	operand 3 write to FPU part 3	4		
WS(fp)	coprocessor wait state	0 - 4		
OR3(fp)	operand 4 read (MSB)	4		
WS(m)	memory wait state	0 - 8		
OW4(fp)	operand 4 write (MSB) to FPU part 4	4		
WS(fp)	coprocessor wait state	0 - 4		
OAW1(fp)	operand address 1 write to FPU	4 - 8		
WS(fp)	coprocessor wait state	0 - 4		
OAW2(fp)	operand address 2 write to FPU	4 - 8		
WS(fp)	coprocessor wait state	0 - 4		
SYNC(fp)	synchronization busy/wait on FPU	0	3 - 5	
IOC(fp)	input operand conversion to internal format	incl below	16 - 46	incl below
AC(fp)	arithmetic calculation in FPU E - unit	35 - 2200	2 - 700	3 - 19
OOC(fp)	output operand conversion	incl above	38 - 80	3
ORE(fp)	output operand rounding & exception	incl above	6 - 60	

The architecture is pipelined, so the cycles shown as "(1)" for SPUR are already accounted for in the fetch cycles. If an entry in the table is blank, that means it is not applicable to that particular implementation. Otherwise, some number or range of cycles is given to indicate the duration of the event. The floating-point execution is the last four events in this table. *SPUR requires two instructions to implement a floating-point multiply with one operand in memory: a floating-point load followed by the FMUL. Thus, there are two instructions fetches for this sequence.

RR(fp) — interrogate FPU for status

In the commercial systems, the FPU decodes the command or instruction it receives and *then* signals the CPU what to do next — operand transfers and so forth. The CPU must find out what it can do next from the FPU, by way of some response: either testing a signal or reading a register. In SPUR, only register-register operations are allowed, and no other interaction with the CPU is needed before the FPU begins an operation.

IAWx(fp), OAWx(fp) — instruction address (segment or offset) write to FPU

The Intel CPU does not save the operand address or the program counter of the current floating-point instruction. Consequently, both must be transferred to the FPU to allow exception processing. These require two bus cycles each, since addresses are 24 bits and the data path between the FPU and memory is 16 bits wide. Motorola transfers the program counter of the current floating-point instruction only on those instructions that can result in an exceptional condition that might need subsequent service. Since it has a 32-bit path to the FPU, only one bus cycle is needed to transfer it. Motorola does not transfer the operand address. The SPUR architecture automatically saves the PC for all floating-point operations. This obviates the need to transfer the instruction address to the FPU and actually provides for more efficient exception processing, since it need not be read at a later time if necessary, as with Intel and Motorola. Also, since only register-register operations can cause floating-point exceptions, there is no operand address to save, eliminating a series of transfers between the CPU and FPU as is the case with Intel and Motorola.

ORx(fp), OWx(fp) — operand read

On operand transfers, the Intel CPU acts like a DMA controller between memory and the FPU. Double-precision data transfers requite four bus cycles per operand over the 16-bit bus. In contrast, Motorola uses regular bus cycles for all memory transfers and actually reads the operands (32-bits at a time) into CPU registers, and then writes the same data to the FPU, requiring two bus cycles for each half of a 64-bit value, for a total of four bus cycles. The SPUR architecture decouples the operand loads from the arithmetic instructions. The *LD_DBL* instruction is a single-cycle instruction, but due to the latency of the memory system and the absence of forwarding logic [Chen85] on the FPU, the FPU execution unit can not begin the *FMUL* operation until the third cycle after the load is fetched. Since the fetch of *FMUL* consumes one of those cycles, only one cycle is potentially lost due to the stall.

IOC(fp), OOC(fp) — operand conversion to/from internal format

All FPU's in this study operate on extended format data (more than 64 bits for exponent and data — see Appendix A for a discussion). Single-precision and double-precision operands transferred from memory must first be converted to/from this internal format before being stored in the FPU register file or memory. The SPUR FPU does this automatically in a single cycle on loads and explicitly with either the CVTS or CVTD instruction before stores. The microcoded Intel and Motorola FPU's use several cycles to make the conversion both on input and output.

AC(fp) — arithmetic calculation in FPU E-unit

Finally, the operation time, or number of cycles consumed by the FPU execution unit in performing the arithmetic is accounted for.

ORE(fp) — operand rounding and exception handling as specified

Since internal operations are done in extended format, transfers to memory require that the operand be rounded to the specified precision before transfer. The SPUR FPU hardware does this automatically in the final phase of calculation. The microcoded Intel and Motorola FPU's use several cycles to round to the extended format (the default), and many times more for rounding to single and double precision formats.

From our previous definition of overhead (any time the coprocessor execution unit is idle) and this discussion about the hardware protocols of each of the FPU's, we can identify the operation and overhead portions of the execution time illustrated in Figure 4-5. It is possible to identify groups of the factors in Table 4-9 and show how implementation details affect these groups, and hence, the system performance. Table 4-10 and Figure 4-7 show the relative fraction of time spent in operation versus time

spent in overhead for each of the three systems and programs.

Table 4-10. Fraction of Execution FPU Busy versus Overhead

Program Version	Param-eter	Intel			Motorola			SPUR		
		GE	DP	PE	GE	DP	PE	GE	DP	PE
COMP	FPU Busy	0.478	0.481	0.485	0.510	0.456	0.625	0.737	0.647	1.000
	Overhead	0.522	0.519	0.515	0.490	0.544	0.375	0.263	0.353	0.000
ASSM	FPU Busy	0.540	0.559	0.542	0.568	0.593	0.667	0.875	1.000	1.000
	Overhead	0.460	0.441	0.458	0.432	0.407	0.333	0.125	0.000	0.000

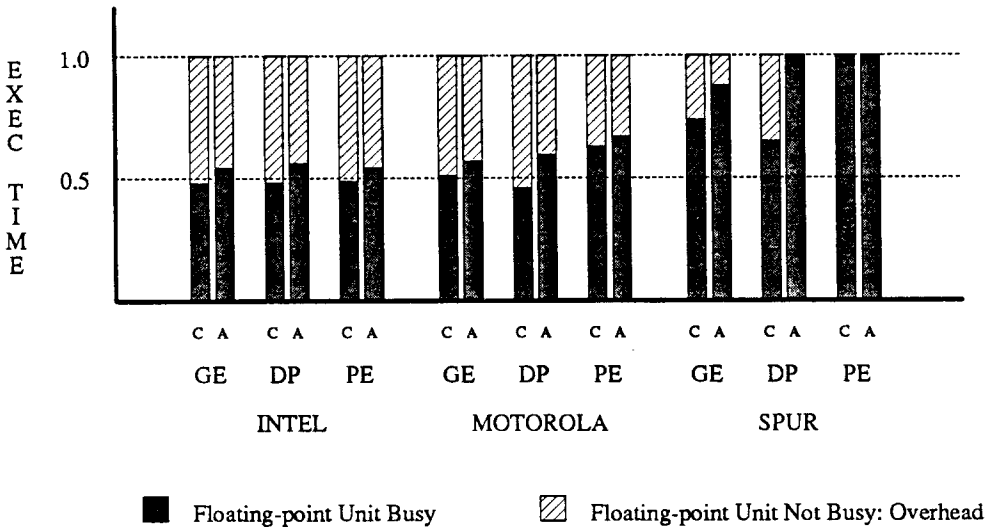


Figure 4-7. Floating-point Operation Time vs. Overhead Time.

This figure plots the data in Table 4-10 showing how much of the total execution time the FPU is actually busy in computation. The rest of the time, with respect to the FPU, is overhead.

Several things are evident from Figure 4-7. First, in contrast to Figure 4-6 which differentiated between floating-point and non-floating-point execution time, Figure 4-7 shows that the commercial FPU's are busy only about half the total time when considering the overhead factors more closely. This represents a significantly different picture in terms of hardware utilization and effectiveness. Second, from this data, one could conclude that the operation time and overhead time are relatively *balanced* for Intel and Motorola. But, since each of the CPU-FPU pairs is designed to allow concurrent execution, the goal would be to keep both execution elements as busy as possible, not just balanced. In the ideal case, this should be possible since the CPU and FPU each operate on different data types and each has its own set of registers. In the following section, we examine the value and effectiveness of concurrent execution.

4.4.4. Concurrent Execution

Most commercial coprocessor architectures claim to allow the processor to proceed while the coprocessor continues to execute in parallel. However, in many cases, floating-point instructions have built-in serialization with respect to the main CPU operation. Both the Intel and Motorola processors have control associated with either allowing or not allowing parallelism. The Intel compilers follow most floating-point instructions with an explicit WAIT instruction, stopping the CPU from further execution (including integer instructions) until the coprocessor BUSY signal is unasserted [Kane85]. Likewise, the Motorola coprocessor can prevent parallel execution by explicitly encoding a *CPU busy-wait* request in the floating-point instruction [Moto85, Sarr85].

The SPUR architecture allows full parallelism between the CPU and FPU. The CPU may issue any number of non-floating-point instructions after initiating an FPU instruction. The interface is fully synchronous, avoiding the inefficiencies of asynchronous interaction between the CPU and FPU. The FPU BUSY signal is continuously monitored by the CPU and indicates during the register-write cycle that the FPU is ready to begin another instruction. The interface provides an effective one-instruction queue, such that no execution cycles are lost between back-to-back FPU instructions.

In order to better observe the contribution of concurrent execution to system performance, Table 4-11 shows the percentage of execution time that both the CPU and FPU are busy, as well as the amount of time there is concurrent execution for all programs. To better illustrate the value of concurrent execution, Figure 4-8 illustrates the overlap not shown in Figure 4-7, where the overhead cycles that occurred during FPU-busy time were masked out.

Table 4-11. CPU Busy, FPU Busy, and Overlap of CPU and FPU Operations

Program Version	Param-eter	Intel			Motorola			SPUR		
		GE	DP	PE	GE	DP	PE	GE	DP	PE
COMP	CPU Busy	0.522	0.519	0.515	0.679	0.719	0.678	0.684	0.588	0.636
	FPU Busy	0.478	0.481	0.485	0.510	0.456	0.625	0.737	0.647	1.000
	Overlap	0.000	0.000	0.000	0.189	0.175	0.303	0.421	0.235	0.636
ASSM	CPU Busy	0.589	0.623	0.597	0.643	0.759	0.656	0.625	0.591	0.455
	FPU Busy	0.540	0.559	0.542	0.568	0.593	0.667	0.875	1.000	1.000
	Overlap	0.130	0.183	0.139	0.211	0.352	0.323	0.500	0.591	0.455

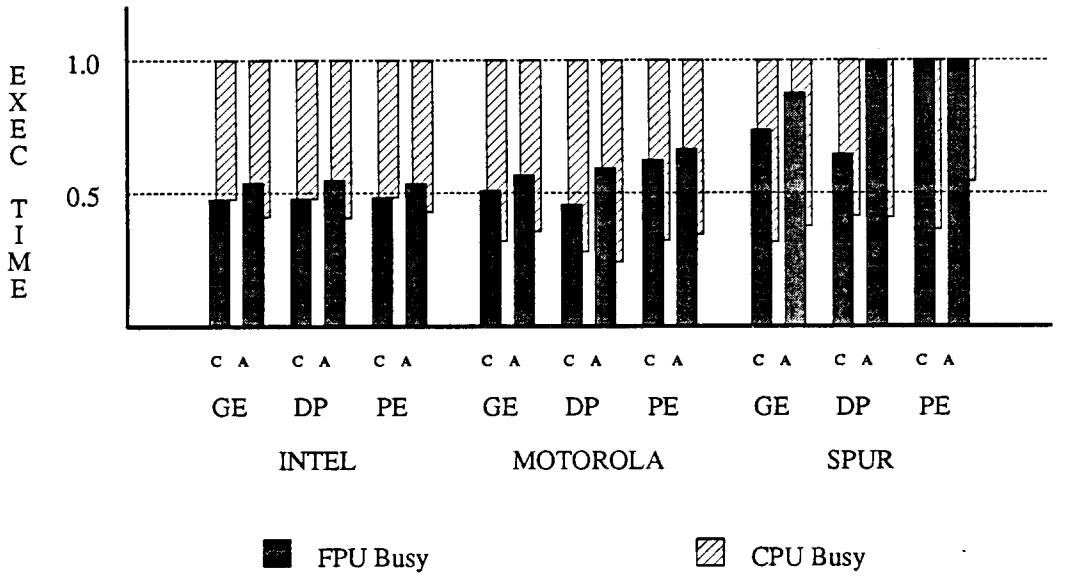


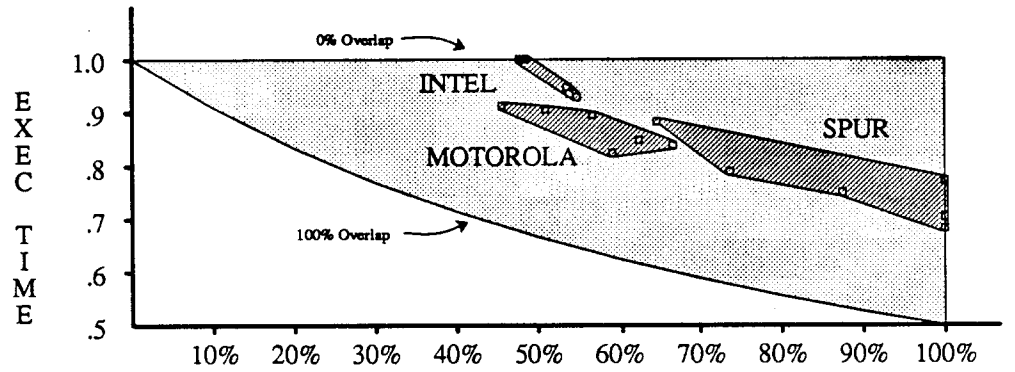
Figure 4-8. Concurrent Execution Between CPU and FPU.

This figure shows the CPU busy time (cross-hatched bars) and the FPU busy time (solid bars) during the total program execution time. The gap between the x-axis and the bottom of the cross-hatched bars represents the aggregate amount of time the CPU is idle waiting for the FPU to complete an operation. Likewise, the gap between the top of the FPU busy bars and the line representing execution time of 1.0 represents the amount of time the PFU is idle during program execution. The region of overlap between CPU busy and FPU busy bars represents the net amount of time that both execution elements are working concurrently, as shown in Table 4-11.

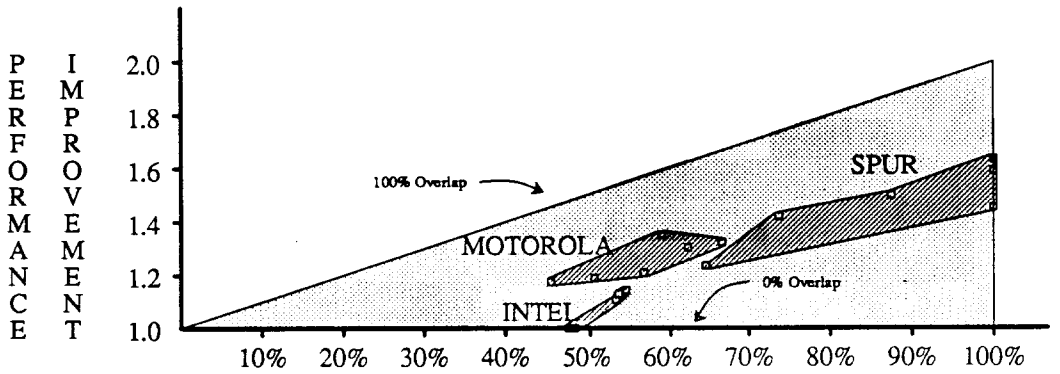
From Figure 4-8, we note that concurrent execution is of benefit to all systems, but in substantially different degrees. For Intel, floating-point operations are relatively slow, and any amount of concurrent execution is small in comparison — as little as none for compiler output and always less than 14% of

total execution time even for hand-coded versions. On the other end of the spectrum is SPUR, where the floating-point operations are much faster. In this case, the loss of performance that would come from non-concurrent execution ranges between 24% and 64%. Motorola is in between the other two — the duration of the floating-point operations is relatively long and the time spent by the CPU in productive effort during concurrent execution is significant. Yet, the operations are not so long that the CPU finishes what it is doing only to have to wait a long time, as is the case with Intel. Consequently, parallel operation allows the system to operate between 18% and 35% faster than non-concurrent operation would allow.

The amount of overlap represents the amount of time saved for program execution in concurrent mode. If we consider *speedup* to be the ratio of execution time for non-concurrent operation to that for concurrent operation, and plot that against the percentage of time the FPU is busy, we observe a measure of the relative value of concurrent operation for each architecture. As illustrated in Figure 4-9, the maximum amount of relative speedup for two execution elements is a factor of two. The execution time as a function of parallelism and resultant speedup are both shown for maximum and minimum overlap.



(a)



(b)

Figure 4-9. Concurrent Execution and Relative FPU Busy Time.

This figure shows how execution time and the relative amount of concurrent operation interact to yield effective speedup for the three CPU-FPU pairs. The lightly shaded region indicates the possible execution times and speedup values as a function of the percentage of time the FPU is busy during the total execution time. The cross-hatched regions indicate the actual execution times and speedup values for individual programs for each of the three systems.

As shown in Figure 4-9, the Intel architecture affords only a small amount of improvement in performance through parallelism. The mechanism for transferring operands is rather sophisticated (similar to a DMA controller) and yet as will be shown in Section 4.4.5 actually results in *reduced* overall performance. Also, since any instructions that have anything to do with floating-point operations or operands are part of the floating-point instruction set, the CPU stalls while any of them are in execution, and some functions which could normally proceed in parallel are *locked out.* Also, the improvements brought about by hand-optimizing the code yield only a modest increase in performance, once again pointing to restrictions in the architecture and implementation to achieving more concurrency.

Motorola is slightly better than Intel. Again, the value of the concurrent execution model is reduced due to inherent serialization of most events associated with floating-point operations or operands. The method of transferring operands in not as complicated as Intel, essentially like programmed I/O, but with the protocol invoked automatically by the hardware, and many of the same restrictions on concurrency. As illustrated by DP ASSM, hand optimization provides more useful FPU cycles as compared to compiled code. This stems from using register-register operations, leaving the partial product in a register on the FPU chip rather than incurring the cost of a memory transfer.

SPUR benefits substantially from concurrent execution. Since the number of cycles spent in either operation or overhead is relatively small, even seemingly small decreases in the number of total execution cycles results in useful increases in performance. By unrolling the loop once, DP illustrates how the concurrent execution model for hand-optimized code can yield a rather impressive speedup of 60%. For PE, SPUR compiled code is hurt more by a sequential model for the hand-optimized code. This comes from the fact that the compiled code loop is longer with more instructions. Consequently, the serialization results in much more loop overhead, that is effectively masked out in concurrent operation. Thus, the hand-optimized code is helped *relatively less* from the concurrent mode.

To summarize, parallel execution involves a complex set of interactions between the components of the system and the software running on the system. Nevertheless, valuable improvements in performance can be attained if (1) the time consumed in floating-point operations is relatively close to the time spent on other operations in the program loop, and (2) the amount of overlap provided by the architecture is significant. Figure 4-10 shows the relative performance improvement in each of the systems as a result of concurrent execution.

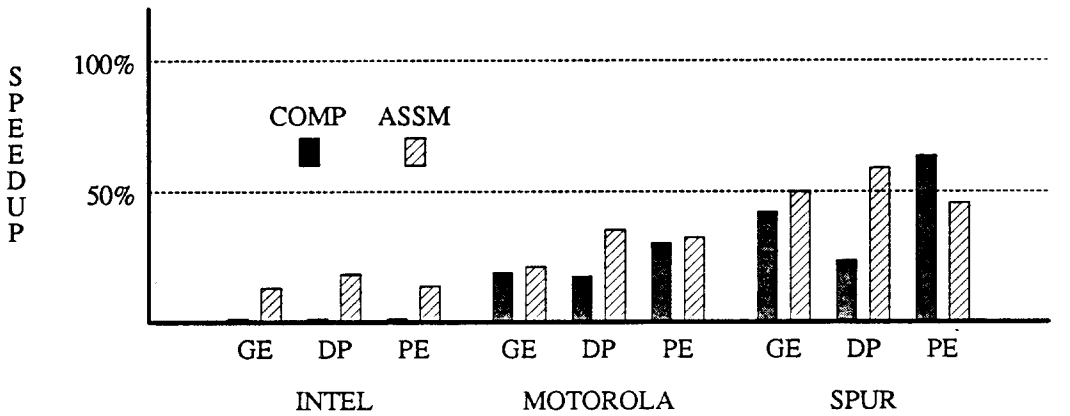


Figure 4-10. Speedup Resulting From Concurrent Execution.

This figure illustrates the improvement in performance afforded by concurrent execution model for the Intel, Motorola, and SPUR CPU-FPU architectures. As illustrated, compiled code for Intel restricts parallelism, while hand-optimized versions run 10% to 14% faster. Motorola speedup ranges between 18% and 35% over sequential execution. SPUR performance improves by nearly 65% in some cases and by at least 24% in others.

The concurrent execution model described in this section would seem to be the obvious choice when considering maximizing system performance. However, there are circumstances when the costs of providing that capability outweigh the benefits. In the following section, we explore the Intel system

to determine cases in which a simpler model would be better.

4.4.5. Performance Degradation Due to Concurrent Execution

As noted in Section 4.2.1, the Intel implementation requires that addresses of floating-point instructions and floating-point operands be transferred to the FPU. This typically requires several bus cycles. If, on the other hand, an execution model were adopted where the CPU would simply stop and wait for the FPU to finish an operation before proceeding (as is done by the DEC MicroVAX 78032/78132 chip-set), how would that affect overall performance? The potential time-saving advantage of this *non-concurrent* execution paradigm would be that the addresses need not be transferred to the FPU, with the commensurate savings in bus cycles expended during the instruction. Table 4-12 shows how performance is actually *decreased* by an average of 15% for the current implementation of the Intel CPU-FPU architecture, due to the amount of overhead incurred in setting up for parallel execution.

Table 4-12. Intel Performance Degradation Due to Concurrent Execution Mechanisms

Program Version	Program		
	GE	DP	PE
COMP	-15.4%	-11.5%	-15.9%
ASSM	-17.7%	-12.7%	-17.5%

This table shows that system performance is adversely affected by the implementation and mechanisms for achieving concurrent operation in the Intel system — an average of more than 15%.

Obviously, this is application dependent. In a situation where more integer instructions could be hidden during floating-point execution, the value of concurrency would be increased. The crossover point is where the cost of setting up concurrent execution is equal to the amount of time to execute other necessary operations in the loop; i.e., integer instructions or cache-miss service. For Intel, the setup for parallel execution is equivalent to approximately 10 to 15 integer instructions. Considering that this is the result of only one floating-point instruction necessitating address transfers, a ratio of 10 to 15 integer instructions per floating-point instruction could be tolerated and still achieve at least the same and possibly better performance, not to mention the advantages of the simpler interface protocol.

From our studies and those of others mentioned earlier, we found that most floating-point applications have tight inner loops, with small numbers of instructions. Consequently, the ability to "hide" integer operations during time-consuming floating-point operations is limited. In reality, although parallel operation can and does improve performance, the most effective means is through more efficient and faster floating-point execution rates, which will be considered in Section 4.5.4. The time/space/technology tradeoffs that were considered in that regard with respect to the SPUR system are discussed in [Bose88b]. In the following section, we will examine how the implementation of each system affects various components of overhead and, as a result, the system performance.

4.5. Implementation Effects on FPU Performance

In Section 4.4.3, we showed that execution time is influenced by many factors. We determined that a large fraction of time is spent in overhead — necessary but largely ineffective cycles. We would like to eliminate them if we can, or at least eliminate their *effect*. On the basis of absolute execution time or comparison of overhead, one might argue that one style of interface is superior to another. However, certain issues incidental to the style of the interface may be more significant in terms of performance than the interface protocols and mechanisms themselves. For example, Intel uses a 16-bit wide data path between memory and the FPU, while Motorola uses 32 bits, and SPUR uses 64 bits. How has that single difference colored our results so far? There are other factors to consider as well, such as memory wait states, implementation mechanisms for synchronization, operation time, and

memory hierarchy latency. We need a fair basis for comparison to better explore floating-point coprocessor efficiency.

In the following sections, we isolate the effects of several features of each implementation to determine the overall effect on the system and provide a better basis for comparison.

4.5.1. Modeling the Systems and Computation

To consider implementation effects on system performance, we developed models of the CPU-FPU architecture for each of the systems. For SPUR, an instruction set simulator [Tayl86] and a functional simulation model using the ENDOT N.2 system and the ISP' hardware description language [Ordy83] were developed. The models yield the number of cycles to perform the various functions, and allow us to control the effects of data bus width between the FPU and memory, operation time, concurrent execution between the CPU and FPU, and cache access time for each of the systems. Table 4-13 indicates the range of the parameters that were varied for each of the models.

Table 4-13. Parameters Varied in Simulation Models

Parameter	Values
Width of the data path for operands transferred between memory (cache) and the FPU:	From 8 to 64 bits, by multiples of 2.
Cache miss service time:	From .3 to 9.6 microseconds, by multiples of 2.
Floating-point instruction operation time:	Factors of .125x, .25x, .5x, 1x, 2x, 4x, and 8x the nominal rates determined by experiment or from data books.
Level of concurrency:	From none (strict sequentiality between the CPU and FPU) to overlapped as much as possible.

This table indicates how the model parameters were varied for our experiment. The data path width represents implementations available in commercial and research machines. Cache service time is largely a function of memory speeds. Also, for multiprocessor systems where contention for shared memory can result wavefront requests [Gibs87] a cache-miss can result in a rather long delay before it receives service — hence, the upper end value. Although scalar FPU's will likely never achieve single-cycle performance for FPU execution for algorithmic reasons, vector architectures currently exist that already provide such. The amount of concurrency was discussed in Section 4.4.4.

To further facilitate our analysis, we make the following groupings of the components of program execution time (referring to Table 4-9):

(1) *floating-point operation overhead*: All cycles associated with the floating-point instruction fetch and data movement between the CPU or memory and the FPU coprocessor are considered floating-point operation overhead cycles. Also included are cycles associated with special functions, such as sending instruction or data addresses to the floating-point coprocessor, testing BUSY, and initiating or transferring the floating-point command. This is designated "FP Ov" in subsequent figures and tables.

(2) *loop overhead*: All cycles associated with incrementing loop counters, performing loop index test and branch, calculating data array addresses, executing integer instructions in general, and performing any no-ops are counted as loop overhead. This is designated "Loop Ov" in subsequent figures and tables.

(3) *memory access overhead*: All cycles associated with the CPU or coprocessor waiting for data to be retrieved from the memory system, including a cache, are considered part of the memory

access overhead. This is designated "Cache Ov" in subsequent figures and tables.

As well as these groupings, we make other assumptions regarding each of the architectures. With respect to fetching instructions, the Motorola CPU has a 2048 bit on-chip instruction cache (64 long words), large enough to accommodate all instructions in small loops. The Intel processor provides an instruction queue of three decoded instructions and a 6-byte deep prefetch buffer. For sequences of integer instructions, often neither Intel nor Motorola expend any cycles fetching instructions. However, branch, jump, call, and return instructions cause the instruction queue of the Intel CPU to be flushed, resulting in instruction fetch cycles until the prefetch buffer and cracked-instruction queue fill again. The SPUR CPU provides a 512-byte on-chip instruction cache (128 instructions). Our simulations indicate an instruction miss rate of 0.0% for the microbenchmarks in our study, once execution has reached steady state.

With respect to operand transfers, each processor is assumed to require no wait-states for memory access. This can only be achieved if an external cache exists. Thus, for this and all subsequent analyses, each system is assumed to have a 128 Kbyte, 32-byte block, direct mapped, mixed instruction and data cache external to the processor. Memory operands are assumed to be double precision floating-point values — eight bytes each — which will cause a worst-case 25% miss ratio on floating-point data accesses based on the assumed linear walk through the arrays held in memory. Data are also assumed to be properly aligned to achieve minimum access time.*

With the models developed and these assumptions, it is possible to analyze the instruction sequences of our microbenchmarks as well as other programs and predict the performance for each system. To validate the projected performance of each of the models, and gain insight into constraints imposed by a particular implementation, the programs were run on workstations using the CPU-FPU pairs in this study. With a logic analyzer [Tekt83, Tekt86], the entirety of each of the loops was captured, recording the operation code, memory references (address and data), floating-point coprocessor status, the user/system modes and control signals. Execution times were measured by timing the benchmarks, and also calculated by counting cycles and multipling by the actual or assumed processor and coprocessor cycle times.

By separating the execution time into its components of overhead and operation, and further, the various components of overhead into memory access overhead, loop overhead, and floating-point operation overhead, it is easier to identify the factors that influence performance the most. Table 4-14 and Figure 4-11 show our baseline comparison and how the normalized and relative execution times shown in Figure 4-5 are separated into floating-point operation and the overhead components. A cache service time of 1.2 microseconds has been chosen for this and following examples. These results represent the commercial implementations, with the bus widths and protocols as described in previous sections.

From Figure 4-11, the dominant overhead factor for the commercial systems and our optimized benchmarks with a 1.2 microsecond cache-miss service time is floating-point operation overhead. SPUR has succeeded in reducing its floating-point operation overhead to between 2% and 17%, while the commercial systems can expect it to be between 25% and 49% of the total execution time, and account far as much as 96% of all the overhead cycles. The main contribution to floating-point operation overhead is the memory traffic, which actually excludes the effects of cache-miss overhead. The SPUR architecture allows parallel loads and stores with floating-point computation which reduces the effect of this component of overhead significantly. Some SPUR CPU-FPU sequences resulted in as little as 2% floating-point operation overhead.

Loop overhead can account for as much as 12.5% of execution time. This represents a relatively small effect, and has been eliminated in certain optimized cases. Normal compiler output would likely result in a higher figure. Some SPUR ASSM language versions of programs are able to reduce loop overhead to zero.

*Alignment on Intel and Motorola architectures is very important. Even with no wait-state memory, non-aligned access on the Intel system can result in 20 extra clock ticks per access, and on Motorola, three times as many bus cycles — 12 clock ticks— as an aligned longword (32-bit) operand access.

Table 4-14. Relative Components of Execution Time for the Nominal Implementation of Each System
(1.2 microsecond cache-miss service time assumed)

Program Version	Param-eter	Intel			Motorola			SPUR		
		GE	DP	PE	GE	DP	PE	GE	DP	PE
COMP	Cache Ov	0.005	0.005	0.003	0.024	0.024	0.024	0.128	0.206	0.088
	Loop Ov	0.048	0.059	0.024	0.102	0.115	0.113	0.125	0.116	0.015
	FP Ov	0.471	0.458	0.490	0.377	0.416	0.254	0.125	0.166	0.034
	FP Oper	0.476	0.478	0.483	0.498	0.445	0.610	0.622	0.512	0.863
ASSM	Cache Ov	0.006	0.007	0.004	0.027	0.044	0.025	0.128	0.234	0.052
	Loop Ov	0.017	0.030	0.022	0.062	0.035	0.070	0.074	0.000	0.010
	FP Ov	0.440	0.407	0.435	0.359	0.354	0.255	0.080	0.056	0.021
	FP Oper	0.537	0.556	0.540	0.553	0.567	0.650	0.718	0.710	0.917

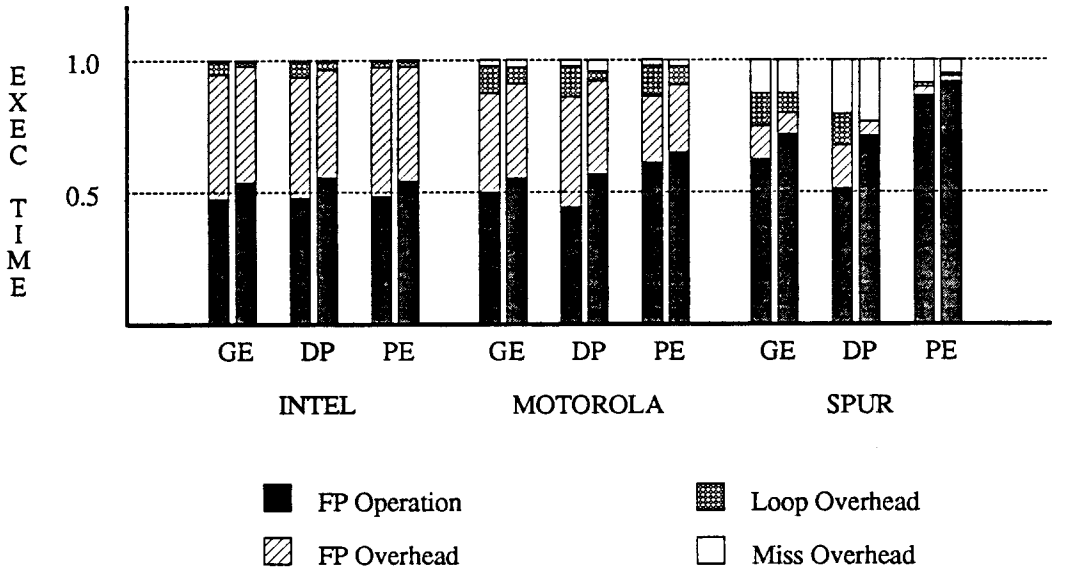


Figure 4-11. Operation and Overhead Components of Execution Time.

This figure shows the fraction of execution time that can be attributed to the various overhead and operation components for each system in its standard implementation as listed in Table 4-14. For this comparison, a cache-service time of 1.2 microseconds has been assumed. See Appendix B for the entire simulation and trace results.

Lastly, the time spent waiting for cache-miss accesses to be resolved is shown as the topmost piece of the bars in Figure 4-11. For commercial architectures, cache access overhead does not amount to more than about 3% of the total execution time. This is simply because the amount of time spent in the floating-point operations and the overhead associated with those operations is so much larger than the cache delay, it is a *relatively* small figure, at least with the assumed cache service time modeled.

However, for the SPUR architecture, since the floating-point operations are fast and loop and floating-point overhead factors can be "buried" during the floating-point execution time, the cache access overhead becomes a significant factor in terms of the amount of time *wasted* per loop iteration — between 9% and 21% of total execution time. Usually, a cache-miss can result in several lost computation cycles. In small loops that consist of just a few operations and associated memory references, the cache can easily become a dominant factor in terms of the time spent in overhead. This is especially true as the cycle time and number of execution cycles per floating-point operation get smaller, as illustrated by the data for SPUR. This all suggests that for newer, high-speed CPU-FPU pairs, one of the more interesting systems issues is the influence of the cache design and implementation on overall performance.

When the number of cycles spent in execution per loop iteration is relatively small, considerable speedup can be obtained by allowing cache access to be overlapped with computation cycles.

Prefetching cache elements during long computation times appears to be a way of saving about 30% of the cost associated with a typical cache-miss for SPUR. Although it is easy to do this with hand-coded sequences, we will have to rely on optimizing compilers if we expect code generated from high-level languages to do this.* From an algorithmic point of view, some loop unrolling and pre-staging of cache accesses can help. This is a topic beyond the scope of this dissertation and will not be considered further.

In summary, reducing the time associated with transferring floating-point operands would have the most beneficial effect on overall performance for the commercial architectures. For SPUR, reducing the miss ratio or discovering ways to hide the cache-miss service time would yield the best improvement in performance. In the following sections, we analyze what happens when each of the architecture models is varied, allowing us to see how changes to the data bus width, cache service time, and operation time affect overall performance.

4.5.2. The Influence of Data Bus Width

In Section 4.4.3 we observed that floating-point operation overhead was the largest overhead component for commercial CPU-FPU systems. Most of the cycles accounted for came as a result of transferring operands between memory and the FPU. The bus bandwidth, which is determined largely by the bus width between the memory and the FPU, will have a large influence on this overhead factor.

To quantify this influence on overall performance, we modeled each of the systems with four data bus widths: 8*,16, 32, and 64 bits. We left the operation times for add, subtract, multiply, and divide as defined by the nominal values for each implementation. The influence of cache-miss service time is independent of the effects seen by changing the data bus width, and consequently was considered zero (it is a separate category and will be covered in Section 4.5.3).

4.5.2.1. The Intel System

In this section, we consider how changes to the bus bandwidth between the CPU and FPU affect the Intel system performance. Table 4-15 is a summary of the results of this experiment, which has been extracted Tables B-1 through B-6 in Appendix B.

The effect of the different size busses on the performance of the system can be observed in two ways. First, in considering the amount of time spent in floating-point operation overhead, increasing the data bus width results in a decrease in the amount of overhead. Conversely, a narrower bus results in more overhead. The percentage change relative to the nominal value is reported in Table 4-15. Second, a side effect of changing bus width and the consequent change in the amount of time spent in floating-point overhead is an increase or decrease in overall performance.

From the tables in Appendix B, nominal floating-point overhead for Intel varies between 41% and 49% of total execution time. This overhead increases dramatically for narrower bus width. By reducing the bus width 50%, overhead increases *more* than 50% — as much as 60% — with an overall increase in execution time of up to 30%. Conversely, doubling the bus width to 32 bits (which is what has been done with the newer i80386/i80387) results in roughly 28% to 35% less overhead and a speedup of at most 18%. Doubling the bus width again to 64 bits only provides an additional 4% to 6% improvement in performance. Careful assembly language programming was only able to reduce floating-point overhead by 1% to 7% compared to COMP versions. Figure 4-12 illustrates both the changes in floating-point overhead and the relative execution time for the Intel system as a function of bus width for the data in Table 4-15.

---

*Although some experts feel that the quality of code from contemporary compilers is no better, and in many cases is significantly worse, than what was available a decade ago [Corb88, Henr88], others are convinced that compiler technology *is* able to meet the challenge of high-speed, pipelined execution models [Henn85, Wulf88].

*Some versions of the IBM PC use an 8-bit bus system with the Intel FPU.

Table 4-15. Intel System Performance as a Function of FPU-Memory Bus Width

Program	Program Version:	COMP			ASSM		
	Bus Width (bits):	8	32	64	8	32	64
GE	% Exec Time for FP Ov	58.4%	39.2%	36.3%	57.2%	34.4%	30.7%
	Change in FP Ov Cycles	56.4%	-28.2%	-36.6%	68.0%	-34.0%	-44.3%
	Speedup (+ by Nominal)	0.79	1.15	1.21	0.77	1.18	1.24
DP	% Exec Time for FP Ov	57.0%	38.0%	35.1%	54.0%	31.3%	29.3%
	Change in FP Ov Cycles	55.8%	-28.0%	-36.5%	68.9%	-34.4%	-40.3%
	Speedup (+ by Nominal)	0.80	1.15	1.20	0.78	1.16	1.20
PE	% Exec Time for FP Ov	60.1%	41.0%	38.1%	56.5%	33.8%	31.8%
	Change in FP Ov Cycles	56.1%	-28.0%	-36.3%	68.1%	-34.0%	-39.7%
	Speedup (+ by Nominal)	0.78	1.16	1.22	0.77	1.17	1.21

This table shows how changing the bus width between the floating-point coprocessor and the memory system affects system performance. As bus width changes, there are three effects to consider: (1) the floating-point overhead as a percentage of execution time, (2) the net change in floating-point overhead, and (3) the resultant change in overall execution time expressed as speedup relative to nominal execution time. The nominal values for floating-point overhead as a percentage of execution time for all programs and versions is given in Appendix B and Table 4-10, and is not included here to make this table slightly simpler.

As an example, let's examine DP COMP. From Table B-1, we find that 2388 of the total of 4968 execution cycles are floating-point overhead in the nominal case. By changing to an 8-bit bus, overhead cycles increase to 3561, representing 57.0% of the total execution time (6248 cycles) and an increase of 55.8% more floating-point overhead than the nominal case. The net result is that the performance decreased 20% since execution time increased nearly 26%. Appendix B contains the simulation results.

By increasing the bus width to 64 bits, slightly more than a 21% improvement in performance is achieved for the Intel system. Although such an improvement is useful, it is important to determine the cost of providing it. If we assume that the on-chip multiplexing necessary to fan-out the 16-bit bus for the double-precision operands is roughly equivalent to what would be necessary for a 32-bit data path, the only effective change would be the number of pads, since the internal data path is 68 bits wide. The nominal i80287 IC has 40 pads in a standard dual in-line package (DIP). Increasing the data bus to 32 bits would increase that to a minimum of 56 pads, but more than likely, a 68-pin standard PGA (pin grid array). The silicon area consumed by the pads and associated drivers is roughly 10% of the chip [Nave80]. Thus, the overall chip area would increase between 3.5% and 7% if all pads were bonded out, and correspondingly, the chip yield would decrease slightly. Nevertheless, the improvement in performance seems justified, if component package type and size are not significantly different. Also, the non-concurrent mode of operation could provide another 10% to 15% improvement, with some small savings in die size due to the elimination of some portions of the control logic needed to support parallel operation. Many other factors would need to be considered in determining if such a change is worthwhile.

In Section 4.5.3.1 and Section 4.5.4.1 we consider the effects of cache service time and floating-point operation time on the Intel system performance. Before doing that, however, we next consider the effects of bus width on the Motorola and SPUR systems.

4.5.2.2. The Motorola System

Analogous to the discussion in Section 4.5.2.1, we present the effects observed by varying the data bus width between the FPU and memory for the Motorola system here. To begin with, Table 4-16 summarizes the experimental results shown in Table B-7 through Table B-12 in Appendix B.

For Motorola, the floating-point overhead varies between 26% and 43% of total execution time when considering all versions of all programs and the nominal bus width (see Tables B-7 through B-12 in Appendix B). Reducing the bus width by 50% to 16 bits, the size of the Intel system, increases the

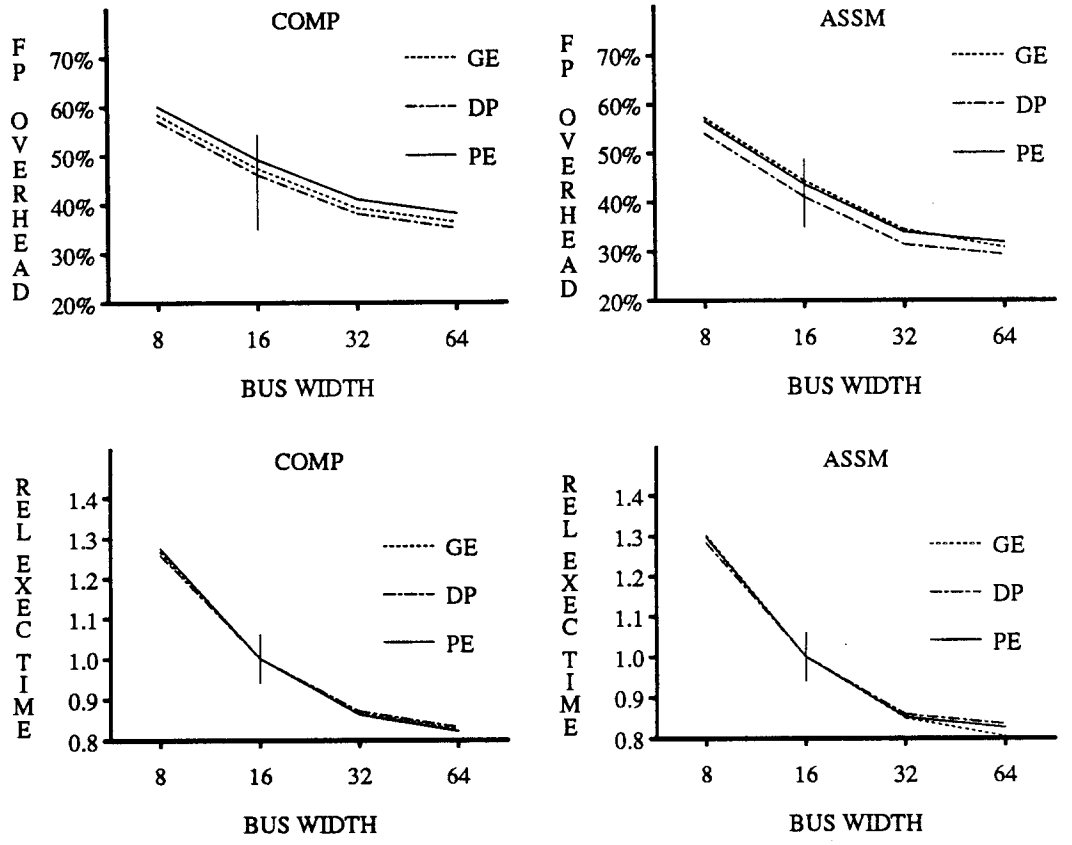


Figure 4-12. Intel FP Overhead and Performance vs. Bus Width.

For each version of each program on the Intel System, these figures illustrate (a) the absolute percentage of execution time that is spent on floating-point operation overhead, and (b) the relative execution time compared to the nominal case. The vertical bar intersects each set of curves at the nominal value for floating-point operation overhead and nominal execution time. As is evident from the closeness of the curves, there is little difference between the different programs.

Table 4-16. Motorola System Performance as a Function of FPU-Memory Bus Width

Program	Program Version:	COMP			ASSM		
	Bus Width (bits):	8	16	64	8	16	64
GE	% Exec Time for FP Ov	61.9%	46.2%	33.0%	60.8%	44.1%	31.8%
	Change in FP Ov Cycles	209.6%	46.0%	-21.3%	214.0%	43.3%	-20.3%
	Speedup (+ by Nominal)	0.52	0.82	1.09	0.52	0.83	1.08
DP	% Exec Time for FP Ov	64.9%	50.2%	36.9%	61.4%	44.1%	32.1%
	Change in FP Ov Cycles	201.3%	45.3%	-21.2%	216.8%	41.9%	-19.6%
	Speedup (+ by Nominal)	0.51	0.81	1.10	0.52	0.84	1.08
PE	% Exec Time for FP Ov	51.8%	31.0%	23.0%	52.9%	31.4%	23.0%
	Change in FP Ov Cycles	229.5%	31.3%	-14.9%	244.0%	33.3%	-15.5%
	Speedup (+ by Nominal)	0.60	0.91	1.04	0.59	0.90	1.04

(Please refer to the caption on Table 4-15 for an explanation of the table entries.)

floating-point overhead by 31% to 46%, or between 31% and 50% of total execution time. This reduces performance and increases execution time by 10% to 19%, depending on the program and version.

On the otherhand, increasing the bus to 64 bits like SPUR results in a performance increase between 4% to 10% and a net decrease in floating-point overhead to about 23% to 37% of overall execution time. Figure 4-13 shows the influence of bus width on floating-point overhead and resulting performance changes. It is apparent that fewer memory references (as is the case with PE) result in

substantially less overhead associated with floating-point operations. In such cases, the width of the path between the FPU and memory has less influence on performance — both positively and negatively. A narrower bus does not degrade performance as much and a any improvements gained by a wider bus width are lessened.

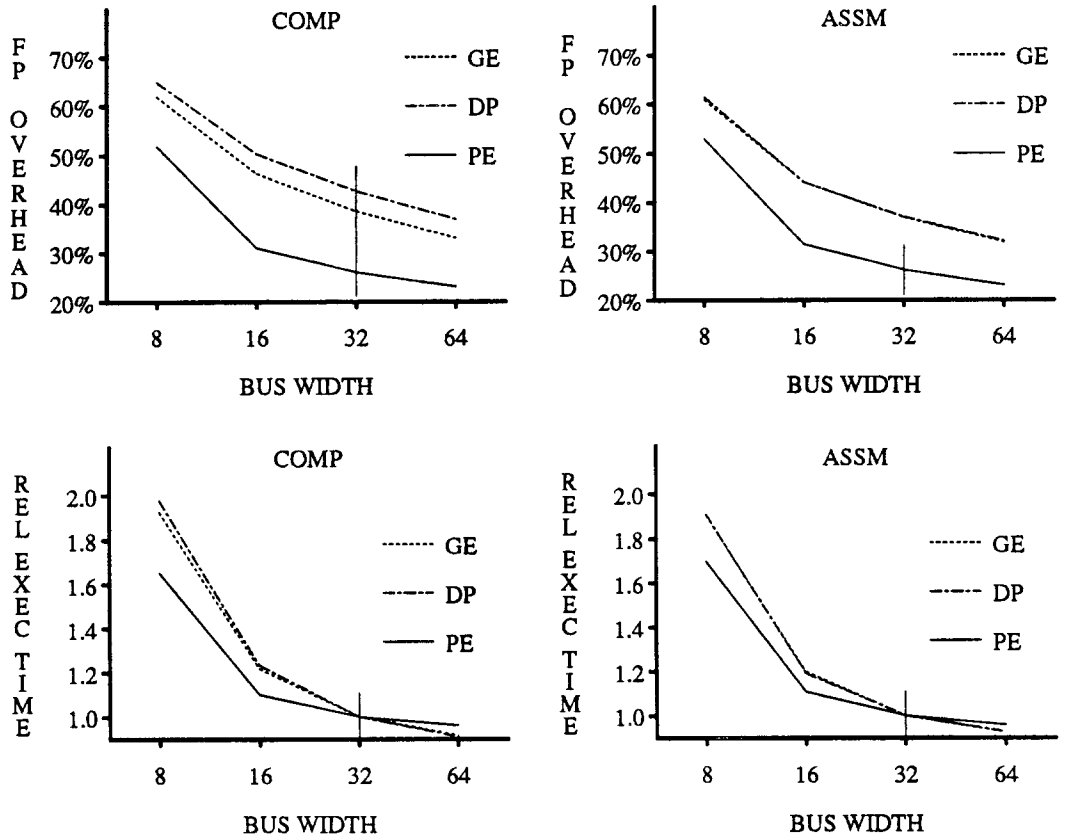


Figure 4-13. Motorola FP Overhead and Performance vs. Bus Width.

As with Figure 4-12, this shows how data bus width between the FPU and memory, floating-point operation overhead, and relative execution time are interrelated for the Motorola system. By doubling the bus width, less than 10% improvement in overall performance results. The vertical bar intersects each set of curves at the nominal value for floating-point operation overhead and nominal execution time.

The MC68881 is packaged in a standard 68-pin PGA. The internal data path is wide — 67 bits — and is multiplexed for the fraction and exponent calculations. Increasing the bus width between the FPCP and memory to 64 bits would increase that to 96 pins. Again, since the area consumed by the pads and associated drivers is 7% of the current chip, overall chip area would increase by 3% to 5%. On the other hand, to multiplex a 32-bit path *on-chip* would require the wide interconnection between input pins and the holding register, which would consume a large portion of chip area. The net result would be an increase on the order of 3% in chip area. Thus, the 4% to 10% improvement in performance may well justify it. Naturally, the chip yield would decrease slightly with the larger chip. As with the Intel system, whether that would be a wise decision depends on many factors and the influence on the overall system.

Floating-point overhead varies over a much wider range of values for Motorola than Intel. As can be seen from Figure 4-13, this comes mainly from the PE programs, where far fewer execution cycles have been spent on floating-point overhead as noted earlier. This stems from the fact that PE causes fewer memory references per loop iteration. In the next section, we consider the effects of varying the data bus width on the SPUR system.

4.5.2.3. The SPUR System

As with the Intel and Motorola systems, the influence of bus width on the performance of the SPUR system is summarized in Table 4-17 based on the experimental results shown in Table B-13 through Table B-18 in Appendix B.

Table 4-17. SPUR System Performance as a Function of FPU-Memory Bus Width

Program	Program Version:	COMP			ASSM		
	Bus Width (bits):	8	32	64	8	32	64
GE	% Exec Time for FP Ov	50.7%	34.0%	21.4%	49.2%	29.5%	15.3%
	Change in FP Ov Cycles	633.3%	241.5%	80.6%	1510.7%	543.8%	173.2%
	Speedup (+ by Nominal)	0.53	0.76	0.90	0.48	0.73	0.89
DP	% Exec Time for FP Ov	56.4%	41.3%	28.9%	57.9%	35.2%	17.6%
	Change in FP Ov Cycles	399.3%	171.2%	56.8%	na	na	na
	Speedup (+ by Nominal)	0.55	0.74	0.89	0.39	0.59	0.81
PE	% Exec Time for FP Ov	26.6%	6.3%	0.0%	11.5%	0.0%	0.0%
	Change in FP Ov Cycles	na	na	na	na	na	na
	Speedup (+ by Nominal)	0.69	0.92	1.00	0.85	1.00	1.00

(Please refer to the caption on Table 4-15 for an explanation of the table entries.) Entries marked "na" indicate where the nominal floating-point overhead was zero.

The SPUR system in its nominal form results in floating-point operation overhead of between 0% and 6% for optimized GE and DP, and is completely eliminated for both versions of PE. This is small compared to both Intel and Motorola. By decreasing the bus width to 32 bits, floating-point operation overhead can increase by as much as 173%, but it is still only 0% to 29% of total execution time, with a reduction in performance of 0% to 23%. This shows that the low-cost protocol of transferring operands between the SPUR FPU and memory, and concurrent loads and stores are only slightly affected by a narrower bus width.

By reducing the bus width even further to 16 bits, the size of the Intel system, as much as a factor of 5.5 times more floating-point operation overhead results, but with an average reduction in performance of 23% (varying between 0% and 38% for all but DP ASSM). For some versions of some programs (see PE, for example), the change in bus width even to 16 bits only results in performance reduction of from 0% to 10%, even when considering the extremes of cache service time, as seen in Appendix B. This is simply because there are fewer operand transfers per floating-point operation in PE and consequently it is not as susceptible to the associated overhead. Figure 4-14 illustrates the floating-point overhead and performance changes for SPUR with the various bus widths.

This figure shows how the SPUR system floating-point operation overhead and speedup vary with different bus width and cache service times. The vertical bar intersects each set of curves at the nominal value for floating-point operation overhead and 0.0% performance speedup.

It is clear from Figure 4-14 that floating-point operation overhead, which is largely a function of bus width between the FPU and memory system, has a much wider variation in its contribution to overall execution time than either Intel or Motorola. There are two reasons for this. First, since the operations that give rise to floating-point overhead are decoupled from the arithmetic operations in the SPUR architecture, overhead factors can be more readily "hidden" through the use of compiler optimizations. As a consequence, in some instances nearly zero floating-point operation overhead results. Second, since the SPUR implementation takes far fewer cycles to implement the programs than do either Intel or Motorola, small variations in the number of cycles associated with overhead will appear as significant fractions of the overall execution time giving rise to the large values in Table 4-17. Related to this is the *relatively long* cache service time for SPUR. The faster cache speeds tend to emphasize the effects of the floating-point operation overhead on overall execution, since the cache service time overhead is small. This will be explored more fully in Section 4.5.3. On the otherhand, the performance of the SPUR system holds up comparatively better than the other two architectures over the span of bus widths considered.

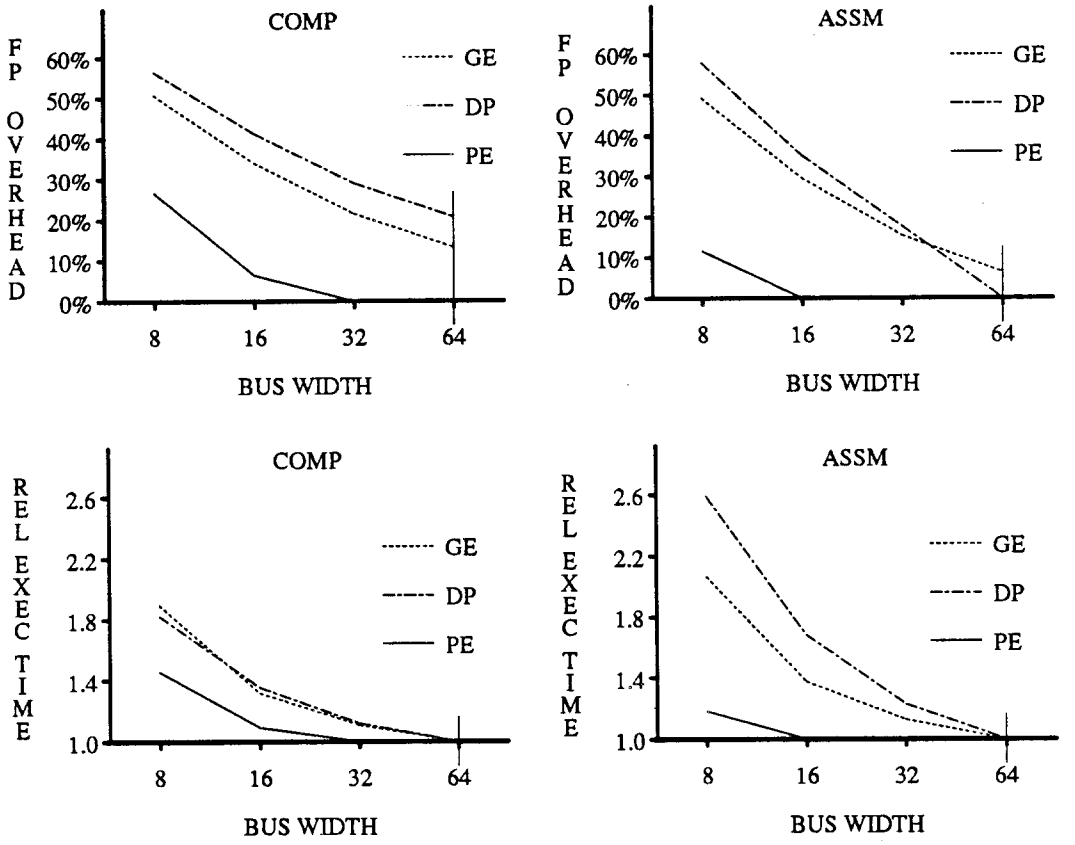


Figure 4-14. SPUR FP Overhead and Performance vs. Bus Width.

Another factor to consider that benefits the SPUR system is the dedicated instruction bus between the CPU and FPU. The commercial architectures rely on passing information in the form of commands between the two processors, which can consume many bus cycles and add to the floating-point overhead figure. Since the SPUR FPU decodes the instructions simultaneously with the CPU, cycles associated with transferring commands or explicit control are eliminated.

The SPUR FPU is packaged in a 208 pin PGA. The useful area of the chip is roughly 11.5 millimeters (452 mils) per side with pad and driver size of 200 by 545 microns. A 32-bit data bus interface would result in a chip area reduction of less than 3%, and slightly more than 3% for a 16-bit bus. These are minor changes in chip area that would result in major changes in overall floating-point performance. When considering double precision floating-point operands and a low-latency interface as with SPUR, the width of the data path is significant.

4.5.2.4. Summary — Data Bus Width

We have explored the interaction of the three floating-point unit interface architectures and several alternative implementations of each architecture. In particular, we looked at the influence of data bus width between the FPU and its memory system for each of the architectures, and the performance effects of different configurations. Although it is unfair to look at absolute execution speed of each of the modified systems — we still see a large difference — if we consider the nominal performance of each of the systems and observe the change in that performance with changing bus width, (also relative to the nominal value) we make a simple relative comparison of each of the systems. Figure 4-15 shows relative performance changes for each of the systems as data bus width is doubled, halved, and so forth.

As illustrated, generally the Intel system is hurt the most by a reduction in bus width. Conversely, increasing the bus width results in a larger percentage gain in performance for Intel than the other systems. This stems largely from the many extra bus cycles needed to transfer instruction and operand addresses to the i80287 FPU to allow concurrent operation, as explained in Section 4.4.5. If that were

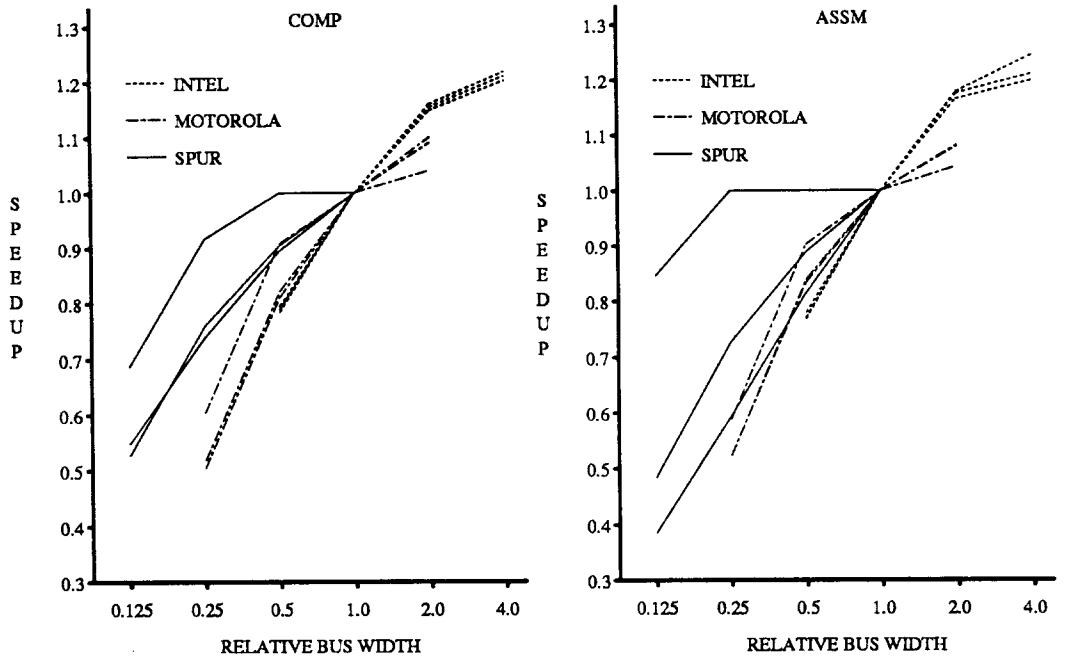


Figure 4-15. Performance as a Function of Bus Width.

This figure summarizes the variation in performance for the Intel, Motorola, and SPUR systems as the relative width of the data bus between the FPU and memory varies.

not the case, relative performance would be more in line with the Motorola system.

Motorola is slightly less affected by changes in the bus width than Intel, but nevertheless, this factor can account for as much as a 20% reduction or 10% gain for a factor of two change.

SPUR is least affected by bus width changes overall. This stems from the fact that bus transactions are relatively low-cost, no matter what the size of bus is. Even when the bus width is cut in half, the running time of some SPUR programs is unchanged. The SPUR architecture encourages use of register variables, allowing the specification of three distinct registers in one instruction. This helps to minimize the amount of memory traffic needed to support typical computation.

From Table 4-3 and [Hans85], we again note that between 0.7 and 1.7 memory references per floating-point operation are typical. This reinforces the fact that memory traffic can become a significant factor in overall performance if the architecture of the system and its interaction with memory are not critically evaluated and mechanisms designed to make that as efficient as possible. We believe that the commercial FPU's suffer as a result of the architecture and implementation constraints we have identified and evaluated. Having considered the various effects of bus width on performance, we now turn our attention to the influence of the cache memory on system performance.

4.5.3. The Influence of Cache Service Time

One of the primary considerations of commercial microprocessor-based systems is the basic cycle time of the memory system. Due to the variability in cost of memory as a function of access time, low-cost systems can be constructed if one is not particularly interested in attaining the absolute maximum performance of the system. Such is usually the case for controller and fixed-function applications, where interaction with low-speed devices may be the norm and only a modicum of processing power is required. In such cases, it would be foolish to provide high-performance memory. On the other hand, running simulations, developing algorithms, and general interactive processing always seem to need more fast cycles than can be provided. In such cases, the cost of the faster-access memory is relatively small when compared with the wasted time of the professionals using slow systems.

Besides speed, many science and engineering applications demand large amounts of memory. Computers have provided users with large virtual memories for years, and many workstation-based systems are no different. As a consequence of the requirements for both fast and large memory, a memory hierarchy is employed in modern systems. It is desirable to have a memory that appears to be as fast as the fastest memory components and yet as large as needed for the advanced applications. The *cache memory* is at or near the apex of this memory hierarchy.

As a consequence of having multiple levels of memory with varying degrees of performance, there may be substantially different latencies associated with memory access, depending on which memory is servicing the request. Obviously, the cache design will influence the latency, depending on such factors as block size, line size, memory technology, replacement strategy, associativity, and so forth. Also, contention for memory resources in a multi-processor system can have some rather long and unpredictable latencies associated with memory access as mentioned in Section 4.5.1. and discussed in [Gibs87]

In this section, we consider the effects of cache-service time on the performance of the floating-point benchmarks run on each of the systems. The model used in this section is based simply on the amount of time that the cache takes to fill a request. Admittedly, this is a somewhat arbitrary means of making a comparison, for each system will be highly dependent on the processor cycle time, memory cycle time, and other factors. Nevertheless, this provides an easy means of differentiating the systems on the basis of this effect and the implementation details need not cloud the analysis.

From the tables in Appendix B, it is readily apparent that the differences in performance as a function of cache service time are quite small for the commercial systems and rather substantial for SPUR. Cache service time plays a significant role in defining overall system performance for SPUR. The cycles spent on servicing the cache are about the same as the number of cycles needed to execute a few floating-point operations. The SPUR FPU implementation consumes only a small number of cycles to perform floating-point instructions in comparison to Intel or Motorola. Consequently, the addition of one or two cycles per loop iteration might easily reduce performance by 10% to 15% for SPUR. Thus, the memory hierarchy design, including the on-chip and external caches, is a prime consideration for current and future single-chip microprocessor systems with single-cycle instruction execution rates. Analogous to the previous discussion about floating-point overhead, Table 4-18 through Table 4-20 and Figure 4-17 through Figure 4-19 illustrate the variations in overhead associated with the time to service cache misses and the corresponding effects on overall performance of the three systems.

Table 4-18. Intel System Performance versus Cache-miss Service Time								
Version	Program	Parameter of Interest	Cache-miss Service Time (nanosec.)					
			0	300	600	1200	2400	4800	9600
COMP	GE	% Exec Time for Cache Ov	0.0%	0.1%	0.2%	0.5%	1.0%	2.0%	3.9%
		Speedup (+ by Nominal)	1.00	1.00	1.00	1.00	0.99	0.98	0.96
	DP	% Exec Time for Cache Ov	0.0%	0.1%	0.2%	0.5%	1.0%	1.9%	3.7%
		Speedup (+ by Nominal)	1.00	1.00	1.00	1.00	0.99	0.98	0.96
	PE	% Exec Time for Cache Ov	0.0%	0.1%	0.1%	0.3%	0.5%	1.0%	2.0%
		Speedup (+ by Nominal)	1.00	1.00	1.00	1.00	0.99	0.99	0.98
ASSM	GE	% Exec Time for Cache Ov	0.0%	0.1%	0.3%	0.6%	1.1%	2.2%	4.3%
		Speedup (+ by Nominal)	1.00	1.00	1.00	0.99	0.99	0.98	0.96
	DP	% Exec Time for Cache Ov	0.0%	0.2%	0.4%	0.7%	1.4%	2.8%	5.4%
		Speedup (+ by Nominal)	1.00	1.00	1.00	0.99	0.99	0.97	0.95
	PE	% Exec Time for Cache Ov	0.0%	0.1%	0.2%	0.4%	0.7%	1.5%	2.9%
		Speedup (+ by Nominal)	1.00	1.00	1.00	1.00	0.99	0.99	0.97

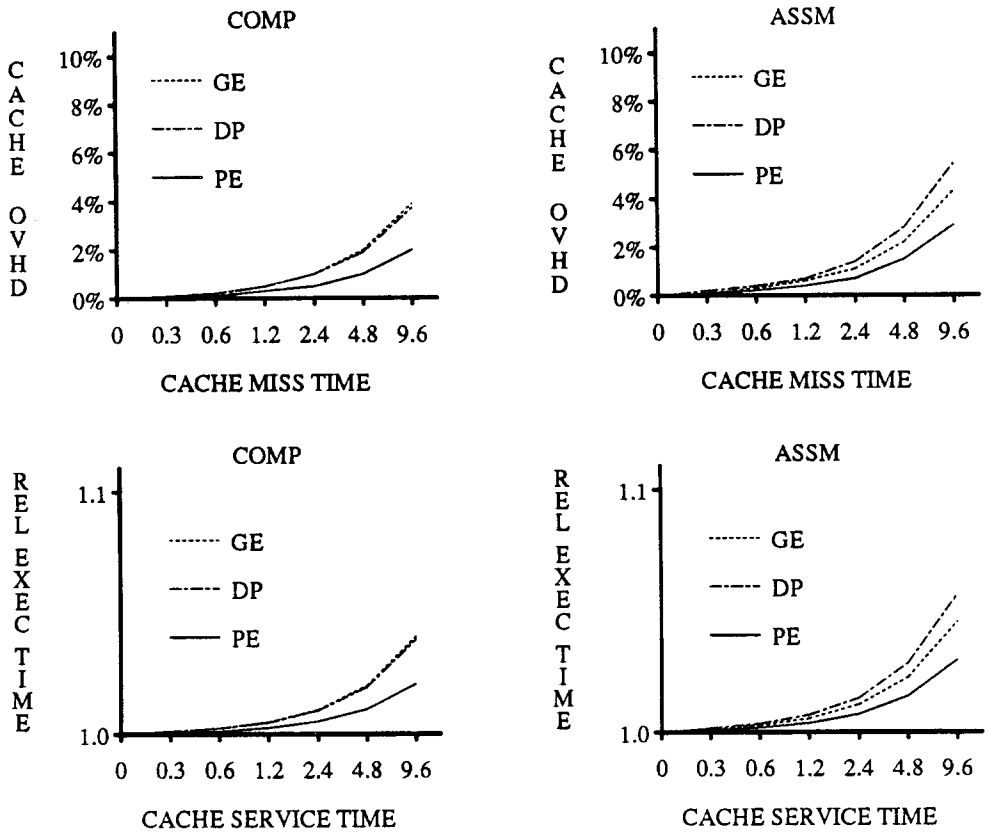


Figure 4-17. Intel FP Overhead and Performance vs. Cache Service Time.

Table 4-19. Motorola System Performance versus Cache-miss Service Time								
Version	Program	Parameter of Interest	Cache-miss Service Time (nanosec.)					
			0	300	600	1200	2400	4800	9600
COMP	GE	% Exec Time for Cache Ov	0.0%	0.6%	1.2%	2.4%	4.7%	9.1%	16.6%
		Speedup (+ by Nominal)	1.00	0.99	0.99	0.98	0.95	0.91	0.83
	DP	% Exec Time for Cache Ov	0.0%	0.6%	1.2%	2.4%	4.8%	9.1%	16.7%
		Speedup (+ by Nominal)	1.00	0.99	0.99	0.98	0.95	0.91	0.83
	PE	% Exec Time for Cache Ov	0.0%	0.6%	1.2%	2.4%	4.6%	8.8%	16.1%
		Speedup (+ by Nominal)	1.00	0.99	0.99	0.98	0.95	0.91	0.84
ASSM	GE	% Exec Time for Cache Ov	0.0%	0.7%	1.4%	2.7%	5.3%	10.0%	18.1%
		Speedup (+ by Nominal)	1.00	0.99	0.99	0.97	0.95	0.90	0.82
	DP	% Exec Time for Cache Ov	0.0%	1.1%	2.2%	4.4%	8.3%	15.4%	26.7%
		Speedup (+ by Nominal)	1.00	0.99	0.98	0.96	0.92	0.85	0.73
	PE	% Exec Time for Cache Ov	0.0%	0.6%	1.3%	2.5%	4.9%	9.3%	17.0%
		Speedup (+ by Nominal)	1.00	0.99	0.99	0.97	0.95	0.91	0.83

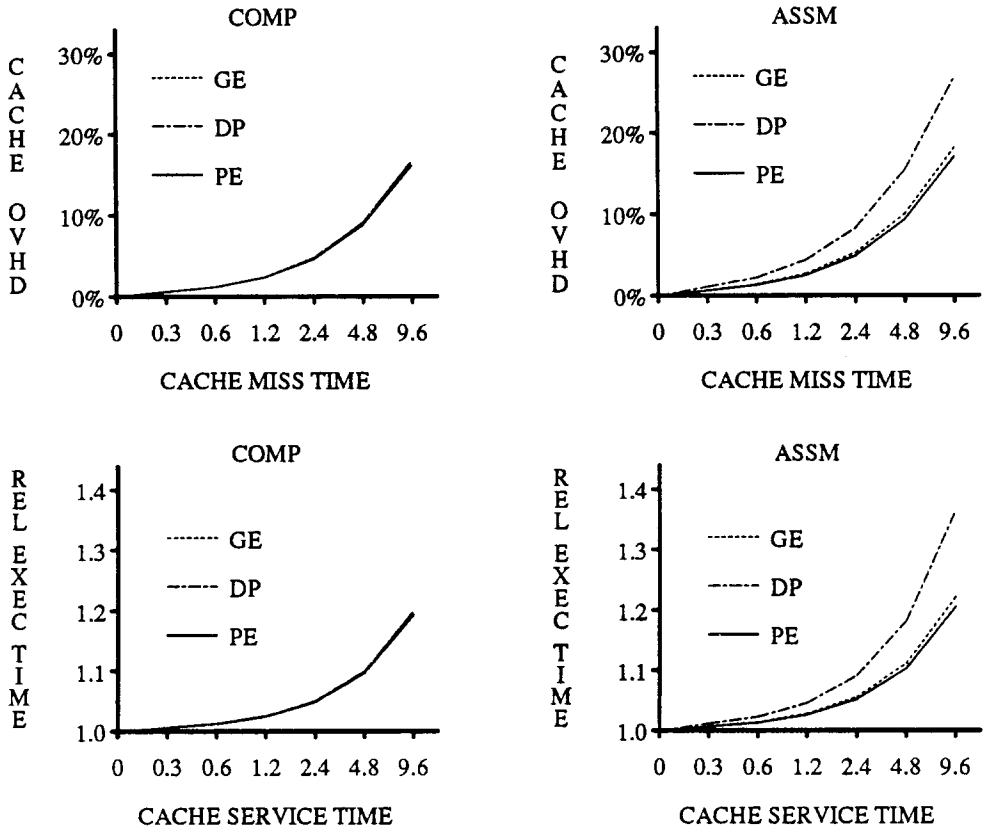


Figure 4-18. Motorola FP Overhead and Performance vs. Cache Service Time.

Table 4-20. SPUR System Performance versus Cache-miss Service Time									
Version	Program	Parameter of Interest	Cache-miss Service Time (nanosec.)						
			0	300	600	1200	2400	4800	9600
COMP	GE	% Exec Time for Cache Ov	0.0%	3.8%	6.1%	12.8%	27.3%	44.7%	62.6%
		Speedup (+ by Nominal)	1.00	0.96	0.93	0.84	0.70	0.54	0.36
	DP	% Exec Time for Cache Ov	0.0%	7.8%	12.5%	20.6%	34.4%	50.5%	66.9%
		Speedup (+ by Nominal)	1.00	0.92	0.87	0.79	0.65	0.49	0.33
	PE	% Exec Time for Cache Ov	0.0%	0.0%	1.6%	8.8%	22.5%	39.6%	58.1%
		Speedup (+ by Nominal)	1.00	0.98	0.94	0.86	0.73	0.57	0.40
ASSM	GE	% Exec Time for Cache Ov	0.0%	2.2%	5.0%	12.8%	29.2%	47.7%	65.7%
		Speedup (+ by Nominal)	1.00	0.96	0.91	0.82	0.67	0.49	0.32
	DP	% Exec Time for Cache Ov	0.0%	5.5%	12.0%	23.4%	40.6%	58.3%	73.9%
		Speedup (+ by Nominal)	1.00	0.88	0.81	0.71	0.55	0.39	0.24
	PE	% Exec Time for Cache Ov	0.0%	0.0%	0.0%	5.2%	20.2%	38.5%	57.9%
		Speedup (+ by Nominal)	1.00	1.00	1.00	0.92	0.77	0.59	0.41

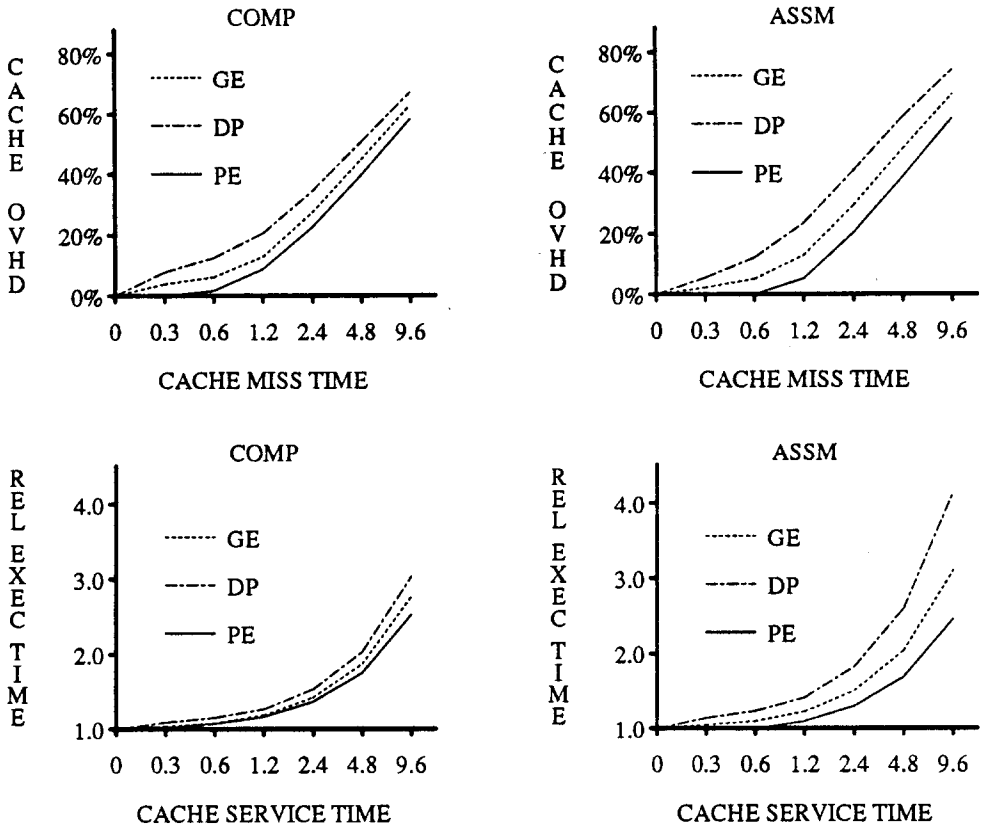


Figure 4-19. SPUR FP Overhead and Performance vs. Cache Service Time.

4.5.3.1. Summary — Cache Service Time

For the commercial systems, cache service time is relatively unimportant. Even for the slowest cache (9.6 microsecond service), the Intel system experiences only a 4% reduction in performance. This simply stems from the fact that everything else in the system takes so many more cycles in proportion; cycles spent waiting for the cache are completely insignificant. For a nominal value of cache time between 1.2 and 2.4 microseconds, the degradation is virtually imperceptible.

For Motorola, the situation is about same. Under typical conditions, usually only a 2% to 5% reduction in performance can be attributed to waiting for the cache. Even with the slowest cache, the loss usually amounts to only about 15%, although in one instance the relative execution time is 37% longer.

As mentioned in Section 4.5.1, a low-latency, single cycle per operation system like SPUR will be affected quite severly by the workings of the cache memory. As listed in Table 4-20 and illustrated in Figure 4-19, a worst-case cache could reduce system performance by 75% compared to the no-miss case. For a nominal cache time of 1.2 to 2.4 microseconds, the performance can be cut nearly in half. It is also generally true that optimized codes are affected more by the cache service time.

The results of this part of our study are based on the most pessimistic assumption — at least one cache miss for every block of operands (a 25% miss ratio). This provides an upper bound on the performance reduction coming from servicing the cache memory. If algorithmic techniques are used to increase the hit ratio, such as strip-mining or performing local-area calculations in total before moving on, significant benefit can result. However, this should not be viewed as a recommended practice. Clearly, keeping the cache service time to a minimum is of the utmost importance to high-speed systems and very likely presents one of the most formidable challenges to the systems designer. We next consider the influence of operation time on system performance.

4.5.4. The Influence of Floating-point Operation Time

The final parameter to consider in our investigation is the actual floating-point operation times of the floating-point instructions themselves. Until now, we have seen how certain features of the architecture and implementation have contributed to or detracted from good system performance. Certainly, a key ingredient is the speed at which the fundamental operations are performed by the floating-point execution unit.

Because there is such a wide variation in the number of cycles used by the FPU's to implement the operations (from as few as three cycles for SPUR's FADD to hundreds of cycles for Intel's FMUL), in this analysis we consider changing the operation times by various factors, as we did in the case of bus width. We vary the time consumed for the fundamental operations between one-eighth and eight times their nominal rate — roughly two orders of magnitude. Tables 4-21 through 4-23 and Figures 4-20 through 4-22 show the results of this part of our experiment.

4.5.4.1. Summary — Operation Time

The Intel system keeps its floating-point unit busy about half the time. In some sense, this reflects a balance with the CPU. On the other hand, it can be argued that roughly half the performance potential is lost. By decreasing the basic operation time by half, the system performance improves by 32% to 37%. This is more than twice the improvement achieved by doubling the bus width to memory. The same ratio holds true for a reduction in performance due to a half-speed execution unit. Thus, first priority for use of resources in a newer version would be spent most profitably to improve the speed of the basic operations. The i80387 does this.

However, as shown in Table 4-21 for a faster FPU, the amount of time that the FPU is busy becomes a smaller fraction of the overall system time. This reflects the performance degradation brought about by the overhead of the interface and data transfer protocols. If the i80287 were sped up by a factor of two, fully 70% of all cycles would be spent in overhead. If it were to run at the speed of the SPUR FPU, well above 90% of all cycles would be consumed in overhead! Clearly, the importance of execution unit speed can not be minimized, but the imposition of factors related to the interface and general architecture of the system can completely eliminate overall cost effectiveness. The system must be viewed and designed as a whole in that regard.

The Motorola system is able to keep its floating-point coprocessor busy between half and two-thirds the time for the nominal system. A 30% performance increase is obtained by doubling the speed of the execution unit. However, the utilization of the FPU drops to 37%. Again, this is an artifact of the overhead inherent in the style and implementation of the coprocessor interface in the system. If the MC68881 speed were comparable to that of SPUR, nearly 90% of all cycles in programs like those tested would be spent with the coprocessor idle. Even though the Motorola system exceeds the absolute performance of the Intel system by a factor of three, it is equally susceptibility to the performance bottlenecks of a cumbersome interface implementation. The MC68882 takes some steps to correct that,

Table 4-21. Intel System Performance versus FP Operation Time

Version	Program	Parameter of Interest	Relative FP Operation Time						
			0.125	0.250	0.500	1.000	2.000	4.000	8.000
COMP	GE	%Exec Time for FP Oper	9.9%	18.5%	31.4%	47.8%	64.7%	78.6%	88.0%
		Speedup (+ by Nominal)	1.73	1.56	1.32	1.00	0.68	0.41	0.23
	DP	%Exec Time for FP Oper	10.2%	18.6%	31.5%	48.1%	64.9%	78.7%	88.1%
		Speedup (+ by Nominal)	1.73	1.57	1.32	1.00	0.68	0.41	0.23
	PE	%Exec Time for FP Oper	10.2%	18.9%	31.9%	48.5%	65.3%	79.0%	88.3%
		Speedup (+ by Nominal)	1.74	1.57	1.32	1.00	0.67	0.41	0.23
ASSM	GE	%Exec Time for FP Oper	12.4%	22.5%	37.0%	54.0%	70.2%	82.5%	90.4%
		Speedup (+ by Nominal)	1.91	1.69	1.37	1.00	0.65	0.38	0.21
	DP	%Exec Time for FP Oper	13.0%	23.1%	37.7%	54.9%	70.9%	83.0%	90.7%
		Speedup (+ by Nominal)	1.93	1.71	1.38	1.00	0.65	0.38	0.21
	PE	%Exec Time for FP Oper	12.2%	22.3%	36.5%	53.6%	69.8%	82.2%	90.2%
		Speedup (+ by Nominal)	1.89	1.68	1.37	1.00	0.65	0.38	0.21

Table 4-22. Motorola System Performance versus FP Operation Time

Version	Program	Parameter of Interest	Relative FP Operation Time						
			0.125	0.250	0.500	1.000	2.000	4.000	8.000
COMP	GE	%Exec Time for FP Oper	9.0%	17.6%	32.6%	51.0%	67.5%	80.6%	89.3%
		Speedup (+ by Nominal)	1.39	1.39	1.27	1.00	0.66	0.40	0.22
	DP	%Exec Time for FP Oper	7.5%	14.7%	28.4%	45.6%	69.5%	77.0%	87.0%
		Speedup (+ by Nominal)	1.30	1.30	1.23	1.00	0.69	0.42	0.24
	PE	%Exec Time for FP Oper	11.3%	21.9%	41.9%	62.5%	76.9%	87.0%	93.0%
		Speedup (+ by Nominal)	1.38	1.38	1.34	1.00	0.62	0.35	0.19
ASSM	GE	%Exec Time for FP Oper	11.3%	16.2%	37.5%	56.8%	72.4%	84.0%	91.3%
		Speedup (+ by Nominal)	1.46	1.15	1.31	1.00	0.64	0.37	0.20
	DP	%Exec Time for FP Oper	9.3%	18.1%	35.9%	59.3%	78.5%	89.3%	90.3%
		Speedup (+ by Nominal)	1.21	1.21	1.20	1.00	0.68	0.39	0.20
	PE	%Exec Time for FP Oper	12.3%	23.9%	45.8%	66.7%	80.0%	88.9%	94.1%
		Speedup (+ by Nominal)	1.41	1.41	1.37	1.00	0.60	0.33	0.18

Table 4-23. SPUR System Performance versus FP Operation Time

Version	Program	Parameter of Interest	Relative FP Operation Time						
			0.125	0.250	0.500	1.000	2.000	4.000	8.000
COMP	GE	%Exec Time for FP Oper	23.1%	30.8%	46.2%	73.7%	93.3%	100.0%	100.0%
		Speedup (+ by Nominal)	1.46	1.46	1.46	1.00	0.63	0.34	0.17
	DP	%Exec Time for FP Oper	20.0%	27.3%	38.5%	64.7%	88.0%	100.0%	100.0%
		Speedup (+ by Nominal)	1.70	1.55	1.31	1.00	0.68	0.39	0.19
	PE	%Exec Time for FP Oper	28.6%	42.9%	71.4%	100.0%	100.0%	100.0%	100.0%
		Speedup (+ by Nominal)	1.57	1.57	1.57	1.00	0.50	0.25	0.13
ASSM	GE	%Exec Time for FP Oper	30.0%	40.0%	60.0%	87.5%	100.0%	100.0%	100.0%
		Speedup (+ by Nominal)	1.60	1.60	1.60	1.00	0.57	0.29	0.14
	DP	%Exec Time for FP Oper	30.8%	42.9%	62.5%	100.0%	100.0%	100.0%	100.0%
		Speedup (+ by Nominal)	1.69	1.57	1.38	1.00	0.50	0.25	0.13
	PE	%Exec Time for FP Oper	40.0%	60.0%	83.3%	100.0%	100.0%	100.0%	100.0%
		Speedup (+ by Nominal)	2.20	2.20	1.83	1.00	0.50	0.25	0.13

as will be mentioned in the chapter summary.

Finally, the SPUR FPU is busy nearly 90% of all available cycles in its nominal configuration. By reducing the speed by one-half, performance would suffer greatly — a 44% reduction. On the other hand, if basic operation speeds increased by a factor of two, better than a 50% increase in overall performance would result, with utilization still at a very high 60%. This illustrates the robustness of the interface design. There is room to accommodate future improvements in technology without unacceptable side effects in terms of wasting resources. By going to essentially a single cycle per operation speed, about an order of magnitude increase, the SPUR FPU would achieve about a 70% performance betterment with the FPU busy only 29% of the time. At that point, other strategies would need to be

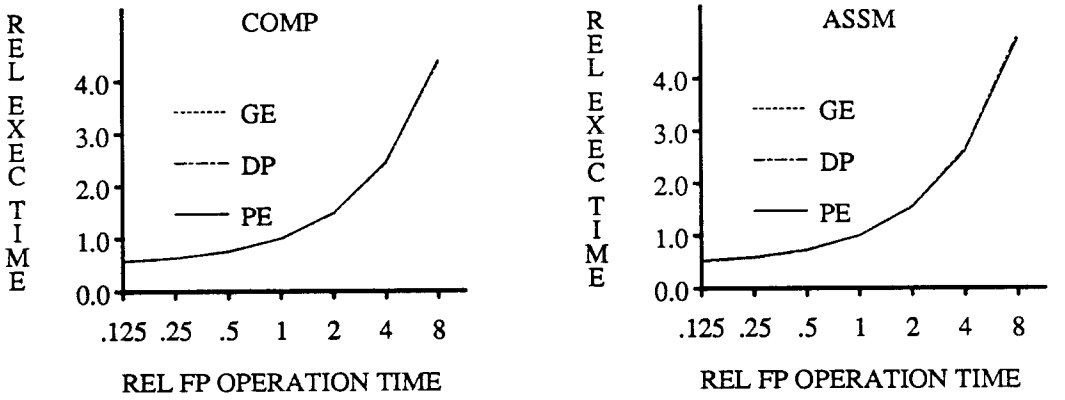


Figure 4-20. Intel FP Performance vs. FP Operation Time.

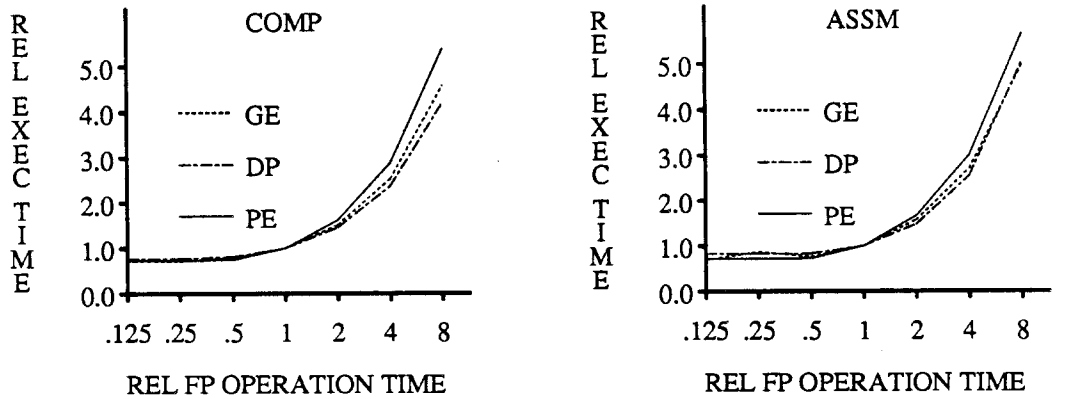


Figure 4-21. Motorola FP Performance vs. FP Operation Time.

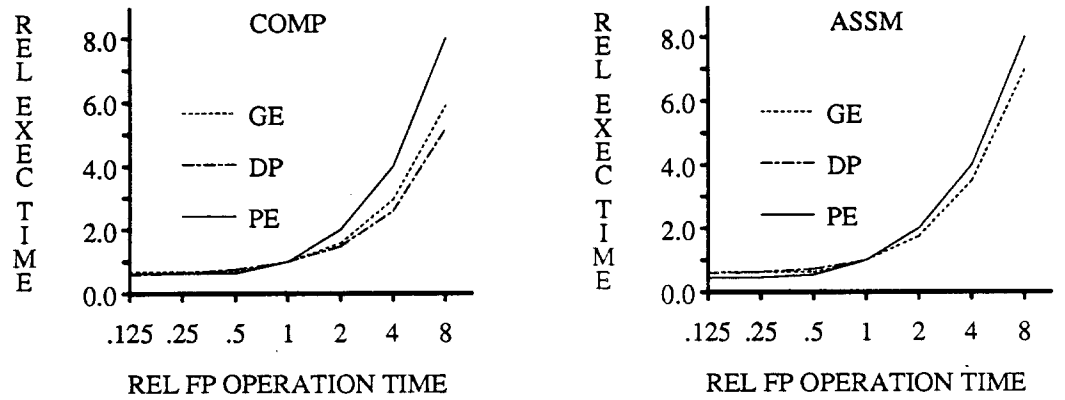


Figure 4-22. SPUR FP Performance vs. FP Operation Time.

employed to achieve the performance potential in such a fast device. Chapter 6 discusses the ideas behind pipelined function units and looks briefly at the architecture of a pipelined floating-point unit.

4.6. Chapter Summary

As time passes, it is inevitable that faster floating point coprocessors will be built. For compatibility reasons, history has shown that current and future products are largely influenced by past ones. Thus, the Intel i80387 coprocessor, even though it has been re-engineered, is a downward-compatible device and complies with the constraints imposed by the general architecture of Intel systems and many of the implementation decisions made in the earlier i8087 and i80287 [Inte87, Stec88]. Although in some cases, it can perform basic operations two to three times faster than the i80287, it is likely, based

on the analyses reported here, that overall performance will still largely be a function of other system and interface parameters.

Likewise, the Motorola MC68882 is compatible with the same family of microprocessors and performs the same functions as the earlier MC68881. For software and many other reasons, commercial systems can not afford to depart from a product once it is established. In the case of the MC68882, the major improvement is in the interface. Rather than couple floating-point operand transfers with the floating-point instruction set (which requires that the CPU stall until a previous operation is done, even if only moving data), there is now a separate part of the interface dedicated to the loading, storing, and conversion of operands. Nevertheless, all transfers still pass through CPU registers, requiring at least twice as many cycles as would be necessary otherwise. This is an artifact of the current implementation, and there is no reason to believe that a future version would not allow direct loads and stores from/to memory. This would naturally require some mechanism to be able to restart an instruction after a memory fault.

As we have seen, if the speed of floating-point coprocessors increases and coprocessor interfaces remain the same, it is possible to achieve minimal overall improvement in performance while letting the fast coprocessor remain mostly idle. This illustrates that current coprocessor interfaces will be less effective as technology provides us with faster components. It also shows that some architectures have a built-in unavoidable overhead that will eventually lead to a substantial limit to performance. This situation leads us to believe that new coprocessor interface architectures will be necessary for future generations of VLSI computers.

We believe that the SPUR floating-point coprocessor and its interface is a step in the right direction. It minimizes interaction with the system and employs efficient algorithms and mechanism to implement the basic functions. It has predictable performance, adheres to the IEEE standard, capitalizes on a synchronous interface with the rest of the system, and maximizes parallelism in many ways. In subsequent chapters, we explore other coprocessor applications to see how well this type and style of interface scale to other applications and processing functions.

<This page is intentionally blank.>

5 Path Optimization Coprocessor Architectures

5.1. Introduction and Overview

There are many problems in engineering, physics, chemistry, economics, operations research, and other fields that deal with *optimization*. In mathematics, optimization considers the problem of finding or approximating a point that gives an optimal (minimal or maximal) value to some function — called the *objective function* — subject to some additional conditions (called *constraints*). Typically, optimization problems fall into one of two broad categories: static optimization (often called mathematical programming) or dynamic optimization. Static optimization is concerned with all forms of time-independent optimization. Dynamic optimization is particularly concerned with models that deal with moving objects where the time variable enters into the optimization. *Dynamic Programming* is related to dynamic optimization and is very effective for handling multistage decision processes [Bell65]. This chapter deals with the application of dynamic programming optimization to path planning and considers coprocessor architectures as alternatives to time consuming software algorithms or expensive, stand-alone hardware implementations.

In Section 5.2 we review some common optimization methods and related applications. The general notions involved in path planning are discussed in Section 5.3, including some broad performance requirements. Section 5.4 considers the direct application of optimization to finding the shortest path, and discusses both brute-force and intelligent algorithms. Section 5.5 outlines the results of our work simulating various algorithms and characterizing their performance. The effects of random data versus typical terrain data sets are then considered, along with data coherence, operand size constraints, and algorithm complexity. Section 5.6 discusses several implementations of the algorithms with emphasis on kernel-function coprocessors. Finally, Section 5.7 discusses the results of our research on path optimization showing that a VLSI path optimization coprocessor can achieve roughly two orders of magnitude improvement in performance over software for a modest cost.

5.2. Optimization Methods

The general problem of solving a system of non-linear equations can often be realized with an equivalent optimization problem using a *least squares solution* [Denn83]. The method of steepest descent, also known as gradient method, uses an iterative algorithm to minimize real valued continuously differentiable functions. Newton's method for solving a system of equations can be modified to an optimization method for certain functions; they must be twice continuously differentiable and the matrix of all second order partial derivatives must be invertible at every iteration point.

Combinatorial analysis is the mathematical study of the arrangement, grouping, ordering, or selection of discrete objects, usually finite in number [Lawl76]. Combinatorial optimization deals with finding the best such arrangement. Some methods of optimization used in various aspects of computer aided design (CAD) work involve the general class of probabilistic hill climbing algorithms [Rome84], for such things as placement of VLSI modules [Caso86, Mitr85], or routing using simulated annealing [Kirk83]. Another application is found in automated stereo perception [Arno83]. Using a modified Viterbi algorithm [Vite79], the maximum similarity path rather than a minimum cost path is determined.

Many problems involve finding the shortest path possible in a directed, acyclic graph from a specified origin to a specified destination. They are often considered to be the most fundamental and important of all combinatorial optimization problems. The application of methods to this problem (path planning and optimization) will serve as a case study for this chapter.

5.3. Path Planning Overview

The problem of path planning can be stated simply: given an environment represented in some form, a start specification and a goal specification, determine a route from start to goal within the bounds of some constraints. For example, given the general environment of the greater Bay Area represented with a road map, start in San Francisco and go to San Jose. Explicit constraints may include travel by automobile and travel time less than two hours. Possible routes are certainly a function of direction (generally, heading south is a good idea), distance (the scenic route may impose a rather excessive average speed), fuel level and consumption rate, time of day (commute-hour grid-lock may make the goal completely impossible within the constraints), weather, roads and freeways available (construction detours need to be avoided), and so forth. Also, other implied constraints may obvious — avoid obstacles, follow roads, no cross-country excursions, and so forth. Determining an *optimum* route is a complex function of all the relevant variables.

The application of computer technology to path planning results from special needs and circumstances — fast-moving autonomous vehicles, indoor and outdoor intelligent robots, dangerous or toxic environments, and so forth. Generally, computerized path planning consists of a hierarchy of tasks. At the highest level, broad goals are specified and general constraints are imposed. Artificial intelligence approaches then use abstract representations of the area of interest and *reason* about objects in that area, determining a sequence of operators to transform an input state space into a goal state. These use the concepts of scripts, plans, goals, semantic meaning, and so forth [Shan77, Wile83].

A lower level in the task hierarchy is the process of determining the actual cost of the shortest or optimum path between two points. The area might be conceptually represented as a two-dimensional grid, with all constraints combined into *directional cost differences*, represented by arcs between adjacent points, that are used to calculate *path length*. The collection of values representing the cost differences are essentially *digital maps* of the area.

This chapter considers the second of these two tasks — computing the shortest path between two points in an area of interest, given the directional cost differences between points in the area. We call this *rectilinear grid path planning*.

5.3.1. General Path Planning Functions

There are many operations involved in calculating the shortest path between two points. These can be grouped into three main functions:

- cost calculation,
- path optimization, and
- path determination.

The cost calculation is concerned with how all relevant variables are combined to produce the directional cost differences for the area of interest. Such factors as distance, elevation, energy consumption, travel time, hazards, and so forth may all contribute to the *cost* of going from one point to another. Figure 5-1 is a simple directed graph representation of a two-dimensional area with *vertices* representing particular points in the area and weighted *edges* representing the costs to travel between points in the area.* We refer to this as directed graph G. Each vertex of G has eight outward-directed edges weighted by the costs to travel to neighboring vertices. These edge weights are the *directional cost data*, and the collection of directional cost data for a given direction (i.e., North, South, East, and so forth) is referred to conceptually as a *directional cost map* (DCM) for that direction. There are eight DCM's, one for each of the points of the compass and the half-angles in between. DCM values can either be precomputed, or computed on-the-fly for certain constrained problems.

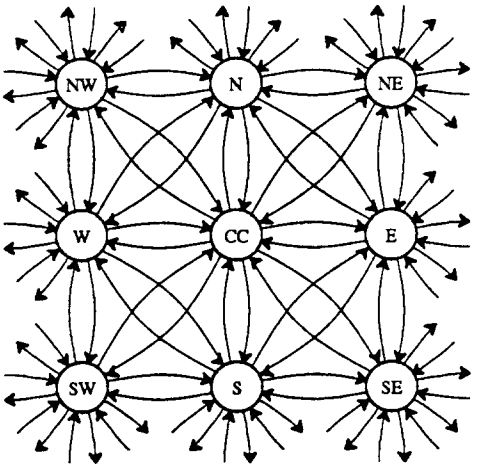


Figure 5-1. Directed Graph Representation of a Two-Dimensional Area.

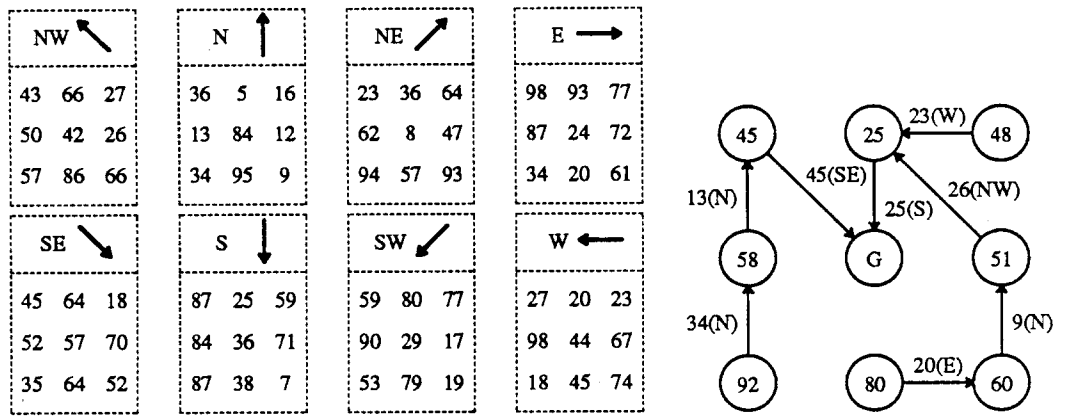
The circles in this figure represent vertices and the arcs represent costs of a directed graph representation of a two-dimensional area. The vertices have been identified by compass points N, NE, E, and so forth, relative to the *center cell*, designated CC. We use the term *cell* interchangeably with *vertex* in reference to rectilinear array-structured graphs.

The second function, path optimization is concerned with finding the minimum cost path from a particular *starting* vertex to a *goal* vertex. The directional cost data provided by the cost calculation function serve as input, and the path optimization process yields a *cost map*, containing the cost from the starting vertex (and probably other vertices) to the goal vertex of G. Some grid path planning algorithms produce a *total cost map* (TCM)** by computing minimum cost *paths* from all vertices to the goal vertex. Each TCM value represents the total cost to travel from that vertex to the goal vertex. Figure 5-2 shows a small portion of a TCM and the DCM's used to compute it.

Once path optimization yields the TCM, the third step in path planning is path determination, which is simply the process of enumerating the minimum cost path from the selected starting vertex to

*We use the term *vertex* interchangeably with grid point and *edge* interchangeably with directional cost, weight, or length.

**We will use the term TCM interchangeably with reference to graph G.



(a) DCM Data (b) Solved TCM

Figure 5-2. DCM Data Sets and Resulting Portion of a Fully-solved TCM.

This figure shows a small part of a total cost map (TCM). Part (a) shows the eight directional cost map (DCM) data sets and Part (b) the TCM with the path from each vertex and the edge costs to the goal vertex shown. Each encircled number represents the total cost to travel from that point in the area to the goal point. The name of the DCM supplying each edge cost value is shown in parentheses.

the goal vertex. This can be done either by (1) finding the DCM value that is equal to the difference between the current TCM value and appropriate TCM value at a neighbor vertex, continuing in a recursive fashion, or (2) retaining the direction of the near-neighbor vertex that is included in the minimum-cost path of the TCM during initial calculations, obviating any further computation to enumerate the path itself. In this latter case, simply following the pointer chain is all that is needed to determine the path. However, extra storage is required to maintain the direction pointers. In any case, the final path may be represented in a vector or an ordered list of vertices, a linked list, or some other suitable structure.

Having identified the three basic functions of path planning and before considering the details of several algorithms involved in implementing path planning functions, we briefly consider overall performance to define the scope of our research.

5.3.2. General Performance Requirements

Let N represent the number of grid points along one dimension of a two-dimensional grid representation. For $N = 512$, the 512-by-512 TCM contains $512^2 = 256K$ elements, and refer to this as a 512-point TCM. For a real-time path planning system, each of the three major functions discussed in Section 5.3.1 must achieve real-time performance. The definition of real-time depends on the resolution of the data sets, the effective rate at which the data sets or goal points change, or on some system-defined update rate. For example, given an initial data set, the minimum-cost path can be determined. Then, if the system is required to continually recompute the minimum-cost path with current position as the starting point, the rate at which the TCM must be updated depends on how quickly the system moves to a neighboring point. If the grid represents a collection of contiguous squares with resolution, say, 10 meters per side, and total cost to the goal is the same everywhere within the square, then a vehicle traveling at the rate of 20 miles per hour must update its TCM once every second. An aircraft may travel at many hundreds of miles per hour through a grid of much courser resolution, and still have a once-per-second update rate. Likewise, if the goal point or any directional cost data change, a new path must be recomputed immediately. For the sake of the work reported here, we assume that solving the TCM in approximately one second defines real-time. This correlates with other contemporary work [Paro85].

Cheng reports that the cost calculation in a uniprocessor software implementation typically requires 15% to 25% of the total path planning time [Chen86], and is repeated each time a constraint changes or new input data are available (resulting in new DCM values). The second function, path optimization, requires 60% to 75% of the total path planning time, and is repeated as often as required, described above. Finally, path determination consumes roughly 1% to 10% of the total path planning time. The time consumed for the entire process can be several minutes, even for small problems, and is to a large degree dependent on either the data (that depends on the algorithms used) or the complexity of the cost-generating function. Thus, uniprocessor software solutions are two to three orders of magnitude too slow.

Multiprocessor implementations of just path optimization using commercially available systems achieve orders of magnitude improvement in performance over sequential computer systems [Lind86]. Nearly linear speedup is reported for various configurations of the Butterfly multiprocessor system, for example. But even the best of these approaches is still about 100 times slower than the one-second elapsed time needed for real-time path planning. Also, such implementations are not very cost effective, considering the large number of processor nodes needed. Table 5-1 reports results of experiments using uniprocessor and multiprocessor configurations for real-time path optimization and is representative of the software state of the art for 512-point TCM's.

Table 5-1. Path Optimization Performance
(512-by-512 TCM, recent software implementations)

CPU System	Convergence Time
DEC VAX 11/780	600 sec.
BBN 5-node Butterfly	510 sec.
Pyramid	300 sec.
BBN 10-node Butterfly	255 sec.
BBN 20-node Butterfly	129 sec.
BBN 40-node Butterfly	63 sec.

This table shows results reported by [Lind86] for software implementations of the path optimization function for a 512-by-512 TCM. These results suggest linear improvement in performance with increasing number of nodes for the Butterfly implementation.

Each of the three path planning functions is a candidate for special-purpose hardware assistance. However, the *cost calculation* can be precomputed for the area of interest. This calculation has inherent parallelism that lends itself to the simple application of multiple processors to improve performance. The *path determination* function is rather simple and demands the least processing time (less than 1% of total time, in many cases). These two functions do not constitute the major portion of the processing time in path planning and consequently are not considered explicitly for coprocessor implementations in this dissertation. Instead, we focus on the most time consuming and computationally intensive aspect of path planning — the *path optimization* function — and architectural issues related to its implementation.

Briefly, results of the studies reported in this dissertation show more than an order of magnitude improvement over the results reported by [Lind86] for software, and substantially more improvement for coprocessor architectures. The improvements come from (1) the efficiencies afforded by clever algorithms and (2) the speed of a specialized but simple hardware coprocessor with function units dedicated to the most often occurring and time consuming portions of the algorithms.

In the next sections, we discuss the basic definitions and concepts involved in path optimization. We describe several algorithms and consider the interaction between the fundamental operations of the algorithms and the underlying data structures used in their implementation. We consider two fundamental techniques used to determine shortest paths — Dijkstra's Algorithm and scan-based algorithms

5.4. Path Optimization

The edge weights of graph G can be a complicated mathematical function of many variables — distance, elevation, time, visibility, fuel consumption, and so forth. But, given those weights, it is a simple matter to determine the length of a path between any two vertices in a directed graph.* Formally, for directed graph G, composed of the set of vertices V and the set of edges E between vertices with weights W, the weight of a path P from v_0 to v_k, including vertices $v_0, v_1, \ldots, v_k$ is the sum of the weights of the edges of P in the graph between v_0 and v_k and is denoted $W(P)$ in Equation (5-1):

$$W(P) = \sum_{i=0}^{k-1} W(\overline{v_i v_{i+1}}) \tag{5-1}$$

As stated earlier, the objective of path optimization is to find the minimum cost to travel from one point in an area of interest — the source — to another point in the same area — the goal. The path from v_0 to v_k is a *shortest path* from v_0 to v_k if there is no other path from v_0 to v_k with lower weight. Finding this path is termed the *shortest-path problem for directed graphs* in the literature [Dijk59].

Lawler [Lawl76] discusses several shortest-path algorithms, including (1) a computational method due to Floyd and Warshall that computes shortest paths between all pairs of nodes, (2) the Dreyfus method, used for finding the *Mth* shortest path between two specified vertices, (3) the Bellman-Ford method of successive approximation or relaxation procedures, and (4) Dijkstra's Algorithm. In general, shortest paths must satisfy Bellman's equations [Bell58].

If we let

$w_{i\,j}$ = the weight (length) of the edge between vertex v_i and v_j if it exists, $+\infty$ otherwise,
l_j = the length of the shortest path from the source vertex v_0 to vertex j.

Then, for a collection of n vertices in directed graph G:

$l_0 = 0$ (the length of the path from vertex v_0 to itself is zero),
l_k = the length of the shortest path from vertex v_0 to vertex v_k, and
$l_j = \min\{l_k + w_{k\,j}\}$ $(k \neq j$, and $j = 1, 2, \ldots,$ n-1). The solution of Bellman's equations for the general case is illustrated in Figure 5-3.

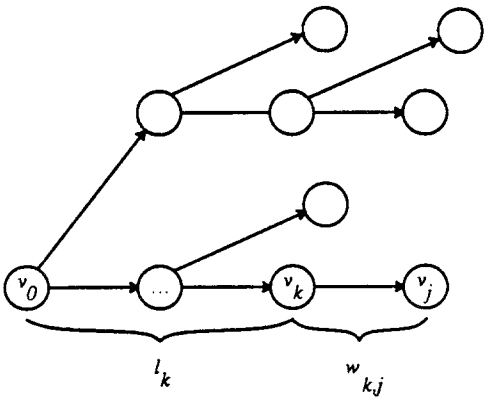


Figure 5-3. Bellman's Shortest Path Tree.

Solving Bellman's equations in general yields an outward directed tree of shortest paths from the source node (vertex v_0) to any arbitrary node. By similarity, the shortest paths from each arbitrary node to a goal node yields an inward directed tree.

In terms of complexity, if all we want to do is determine the sequence of vertices that are contained in a path from the source to the goal, the problem is $O(n_e)$, n_e being the number of edges in G. When finding the shortest path, there is no reason to suppose that more than $O(n_e)$ time should be

*For this and subsequent discussions, we use the terms *cost* and *length* synonymously with edge weight.

required, but no such algorithm is known. It is often the case that finding the shortest path between two *particular* vertices is all that is desired. Nevertheless, no known algorithm is more efficient in the worst case than the best single-source algorithm, Dijkstra's Algorithm for shortest paths, described in the next section.

5.4.1. Dijkstra's Algorithm

Using the nomenclature developed previously, Dijkstra's Algorithm for directed graphs consisting of positive weights between vertices finds shortest paths from the source (vertex v_0) to other vertices in order of increasing distance (cost) from v_0. It stops when it reaches v_g, the goal vertex, or can be modified slightly to find the set of shortest paths from v_0 to every other vertex in the set.

The process is as follows: the value of every vertex in graph G represents a cost and is initially set to infinity (or the largest integer value that can be represented in the computer for software implementations). This represents an *infinite cost* to travel from the source vertex to any other vertex in the graph. The source vertex value is set to zero, representing *no cost* to travel to itself. The source vertex becomes the initial *solution set*, S. The algorithm then finds the next vertex of G that is not in S that forms the next shortest path from the source. This process is repeated and the solution set S grows in an outward-spreading fashion until the specified goal vertex is reached (or all vertices of graph G are included in the solution set S, if the entire set of shortest paths is being calculated). This process is data dependent. If the goal vertex is located next to the source vertex, for example, the first few additions to the solution set — a maximum of eight tries — finds the goal vertex, and the algorithm terminates. A formal statement of Dijkstra's shortest path algorithm follows. The interested reader is referred to [Aho76] or [Baas78] for the proof.

> Given: a directed graph $G = (V, E)$, a source $v_0 \in V$, and a function l for edge weights between vertices constrained to nonnegative reals. Let $l(v_i, v_j)$ be $+\infty$ if $l(v_i, v_j)$ is not an edge, $v_i \neq v_j$, and $l(v, v) = 0$.
>
> Output: For each $v \in V$, the minimum over all paths P from v_0 to v of the sum of the weights of the edges of P.
>
> Method: Construct a solution set S with $S \subseteq V$ such that the shortest path from the source to each vertex v in S lies wholly in S. An array $D[v]$ contains the cost of the current shortest path from v_0 passing only through vertices of S. This approach can be expressed in pseudo-code shown in Figure 5-4.

Although Dijkstra's Algorithm is referred to as a *single source* algorithm, if the sense of direction is reversed, an entirely equivalent algorithm results. In this case, a goal vertex is given rather than the source vertex. Rather than finding the costs from the source to other vertices in the set, the costs *from* all other vertices in the set *to* the goal vertex are found. This modified algorithm operates in the same fashion, with the same complexity, but with the sense of *direction* reversed, and thus becomes a *single sink* algorithm. This modification is useful for path planning situations where the specification of a goal point rather than a starting point is often the case. In the remainder of this dissertation, we assume that paths to a specified goal are to be found.

Dijkstra's Algorithm provides the theoretical and computational basis for the problem of path optimization. In practice, sometimes iterative techniques are used with highly-specialized hardware to do the path optimization function. The following section discusses such *scan-based* algorithms.

The algorithm:

```
    begin
1       S ← v0 ;
2       D[v0] ← 0 ;
3       for each v in V − {v0} do D[v] ← l(v0,v) ;
4       while S ≠ V do
            begin
5               choose a vertex w in V − S such that D[w] is a minimum ;
6               add w to S ;
7               for each v in V − S do
8                   D[v] = MIN(D[v], D[w] + l(w,v))
            end
    end
```

Figure 5-4. Dijkstra's Shortest-path Algorithm.

If V contains n vertices, steps 1-3 are $O(n)$. Step 5 as well as steps 7-8 are $O(n)$. The outer loop, steps 4-8, is $O(n)$, yielding overall algorithm complexity of $O(n^2)$. Alternative search and sort methods enable an $O(lg\ n)$ minimum selection in step 5. This is the key to substantially improved performance. A linear sort is several orders of magnitude less efficient (see Section 5.5.2).

5.4.2. Scan Algorithms

As illustrated in Figure 5-1 above, each vertex has eight neighbors with associated from- and to-neighbor costs, for a total of 16 edges incident to each vertex. To determine the minimum cost for any vertex relative to the goal, eight of these edge-weighted paths and the associated neighbor vertex values need to be evaluated. If the problem is single sink, the incoming edges are the ones of interest, and vice-versa for single source.

For scan-based algorithms, each vertex is considered in turn, with no particular regard to where the source or goal vertices are located. The minimum cost at each vertex is determined by comparing its current value with the eight values calculated by adding the appropriate edge weight to each near-neighbor vertex value. The current vertex value is replaced with the minimum (least cost) value calculated. Then, the *next* vertex is considered. This continues until the values in all vertices *converge* to a final TCM solution (i.e., each vertex has reached its final value and no further changes occur). At this point, the graph G represents a TCM and contains the entire set of shortest paths from all vertices to the goal. During the process, minimum cost values are propagated away from the goal to each cell in the TCM. A *sweep* is defined as one complete iteration where *all vertices* of G are visited at least once.

The graph G is very regular and highly interconnected. Overhead associated with a linked list representation of G or the directional cost maps can be prohibitive in space for large values of N. A scan-based algorithm for determining shortest paths can capitalize on the regular structure of graph G and the DCM data sets. Cheng and Linden [Chen86, Lind86] both describe scan-based, *brute-force* methods to determine minimum cost paths that employ parallel and pipelined techniques. The basic approach for these techniques using the nomenclature of Dijkstra's Algorithm is shown in Figure 5-5. The eight near-neighbors of vertex v are referred to as v_{NNR} in the pseudo-code.

The algorithm:

```
    begin
1       for each v in G do G[v] ← +∞ ;
2       G[v0] ← 0 ;
3       repeat
4           for each vertex v in G do
5               G[v] = MIN(G[v], G[vNNR] + DCM[vNNR])
6       until no more changes in G
    end
```

Figure 5-5. Scan-based Shortest-path Algorithm.

If the TCM contains n grid elements (vertices), step 1 is $O(n)$. Steps 5-6 are $O(n)$ with the outer loop steps 3-6 being $O(n)$ in the worst case, for a total complexity of $O(n^2)$. This compares to Dijkstra's Algorithm of $O\ n(lg\ n)$ for best-case sorting methods.

It is clear that the operations performed at each vertex are the same as those performed in Dijkstra's Algorithm. The main difference is the order in which vertices are considered. Scan-based algorithms simply follow a fixed pattern. For example, by starting with the upper left-hand corner vertex (assuming some rectilinear grid topology), the vertex is *minimized* with respect to its neighbors. Then the neighbor vertex *one step to the right* is considered; and so on, until the *row* is complete. Then the second row starts with the left-most vertex directly below the upper left-hand corner vertex. Again, all vertices in the row are minimized in sequence. Then comes the third row, and so on. The process continues until the very last vertex (the lower right-hand corner) is reached, completing one sweep. This is analogous to the simple raster-scan pattern used to move the beam in electronic video display terminals.

This scanning process leaves several unanswered questions. Is a simple raster scan sufficient? Are there any advantages to scanning in a reverse direction part of the time? Does it matter where the scanning begins relative to the specified goal point? From the literature, we find that the rate at which a solution is achieved is not only data dependent for scan-based algorithms, but also scan-algorithm dependent. Figure 5-6 illustrates several scan-based algorithms that we explore in our research.

Algorithm No. 1 is the simple raster scan described previously. It is probably the most obvious scan-based technique to consider. Algorithm No. 2 modifies No. 1 slightly by starting every other sweep at the bottom of the array instead of the top. This is done to eliminate the potential bias of sweeping only in one direction. Algorithm No. 3 is like No. 2, but scans each row twice — once forward and once in reverse. This technique is evaluated to determine if immediate feedback has a positive influence on convergence rate that justifies doubling the amount of work per sweep (twice as many cell checks per row than the previous algorithms). Algorithm No. 4 is a variation of No. 2, where every other sweep begins at the bottom but scans in the opposite direction. This method determines if the single scan direction causes slower convergence, and yet reduce the amount of work that is needed with No. 3.

Algorithm No. 5 modifies Algorithm No. 3 slightly by using boustrophedonical scans, alternatingly starting at the top and at the bottom. This results in an equal number of cell checks per sweep as the other scanning algorithms, but eliminates the potential single scan-direction bias. Algorithm No. 6 is a variation of No. 4, but flipping the starting point corners for each group of two sweeps to determine if convergence rate is affected. Algorithm No. 7 is the basis of the work discussed in [Lind86] and scans in all four directions. It uses two starting points, the upper left and lower right corners, to begin each pair of scans. Scanning top-to-bottom and bottom-to-top completely eliminates any bias that comes from the strict left-to-right or right-to-left scanning of all previous algorithms and is evaluated to

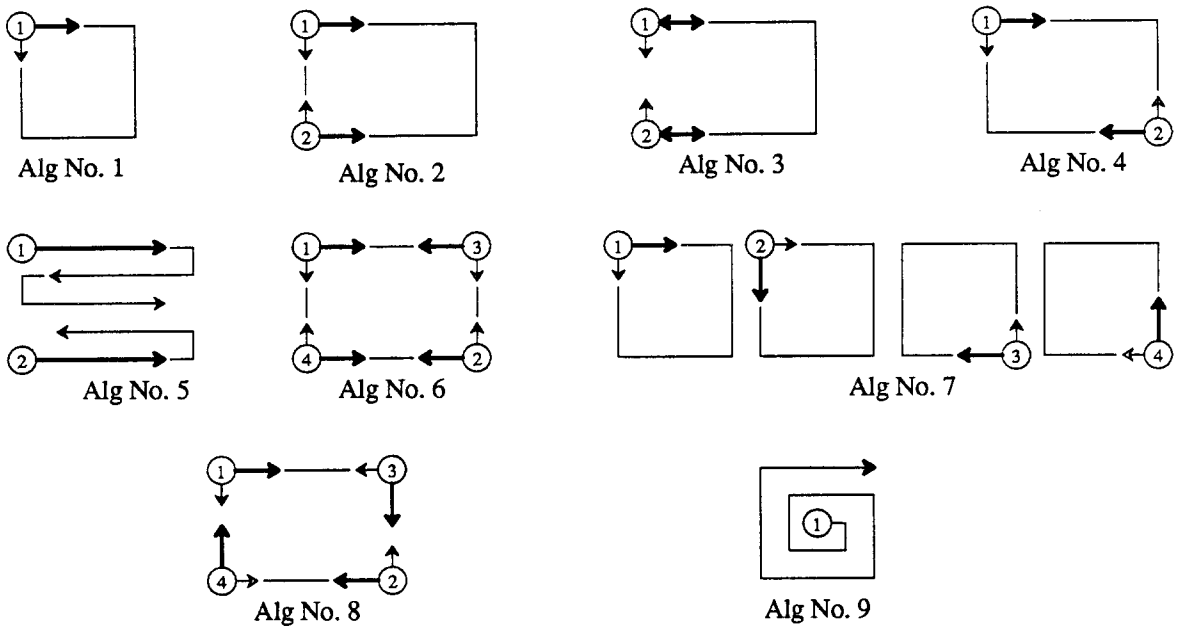


Figure 5-6. Scan-based and Other Sweeping Techniques for Path Optimization.

This figure illustrates the various scanning techniques. The sweep number in sequence is shown inside the small circle, with the inner loop scanning direction designated with the darker arrow, and the outer loop sweep direction with the lighter arrow. For example, Algorithm No. 7 scans four different directions, the first of which is like a video display: raster scanning left-to-right and sweeping top-to-bottom.

quantify that effect, if observable. Algorithm No. 8 is a combination of No. 4 and No. 7 and uses all four corners as sweep start points, alternately choosing between top and bottom as well as right and left corners. Again, we consider this technique to determine if No. 7 is less efficient because it uses only two starting points. With Algorithm No. 9, strict raster-like scanning is no longer used. In this case, a *spiraling out* process around the goal point drifts in an outward direction in concentric circles. This technique should provide the most efficient means of spreading minimum-cost information quickly, since the most recently minimized values will be *pulled* along in the direction of the spiral.*

The process of finding a minimum cost path is also data dependent for scan-based algorithms. Usually, many sweeps are necessary to converge. However, if the DCM data are the same for all directions, any scan algorithm would require only a few sweeps to *find* the solution. Also, different scan-based techniques may have different convergence rates independent of input data. One would intuitively expect to propagate the minimum cost values toward the edges of the graph, away from the goal vertex, by scanning in the four possible directions: left to right, right to left, top to bottom, and bottom to top. However, as will be shown in Section 5.5, the direction of the scan is not the primary factor in determining convergence. Rather, it is the direction that minimum-cost information is *pushed or pulled* during the calculation that allows convergence. The scan direction merely facilitates this process. Also, to converge, some scan-based algorithms require more than one sweep without any changes. This stems from the fact that the minimum-cost information propagates in a direction influenced by the scanning technique. It is possible that a vertex value that changes on one sweep will not influence some other vertex to change until a subsequent sweep, and not necessarily the next one. Thus, we define a *macro sweep* as the minimum number of sweeps needed to guarantee that the graph G has in fact converged. More will be said about this in Section 5.5.

*It is possible to reduce the number of near-neighbor checks necessary for a complete cell-check. This is an optimization that can be applied to several of the algorithms and will be discussed in Section 5.6.2.

We next consider the representation of the graph G in memory and how different data structures influence algorithm efficiency.

5.4.3. Data Structures

Published implementations of shortest path algorithms sometimes presume that the collection of vertices is small and not necessarily regular. Consequently, linked-list data structures are often suggested for maintenance of the graph G. In this case, vertices are divided into two non-overlapping sets: those that are members of the solution set, and those that are not. An *adjacency list* or similar structure is maintained to describe the near-neighbor relationships between vertices in the solution set S and those that are candidates for inclusion in the solution set.

For the problems considered here, consisting of large numbers of contiguous interconnected vertices, two dimensional arrays are a more natural and space efficient data structure. The edges between vertices are implied by the near-neighbors of the array structure. Vertices which are not adjacent in the array have no edges (thus, infinite cost) between them — hence, are not included in the DCM data sets. The arrays are fully populated and have a one-to-one correspondence with the two-dimensional area they represent.

For either Dijkstra's Algorithm or scan-based techniques, one possible data structure providing the advantages of arrays and maintaining locality for good cache performance consists of an array of records. Each record in the data structure contains a TCM current minimum value and the eight associated DCM values. For scan based algorithms, a minimum_direction pointer to the near-neighbor TCM value that is contained in the minimum cost path might also be included.* For Dijkstra's Algorithm, an index into a sorting array is needed. This sorting array is a priority queue, and is used to maintain the list of cells eligible for comparison to determine the next TCM element to enter into the shortest path solution set. The queue itself is maintained as a linear array in memory to allow easy random access based on the index, and uses a heapsort algorithm to maintain the priority order.** Special values (0 and 1) can be used to flag TCM values that are already included in the queue and/or solution set.

For some highly pipelined hardware implementations of scan-based algorithms, separate memories for each of the DCM data sets is imperative to achieve the high-bandwidth I/O needed to support their architectures. In these cases, the dedicated memories are not likely to be part of a general purpose computer system and would not suffer from delays associated with page faults, thrashing, access conflict, or other system related problems. More about the data structures needed for specific implementations is contained in Section 5.6.2.

For both Dijkstra's Algorithm and any scan-based technique, the basic operations and how they relate to an array data structure are illustrated in Figure 5-7. The graph center cell (CC) is minimized by considering its neighbors and the costs of traveling to or from its neighbor vertices. In this case, the DCM data relating the travel costs from the CC to its neighbors are represented by the arrows in Figure 5-7.

TCM near-neighbors and DCM maps are given names matching the compass directions. The neighbor cell indices for both the TCM and DCM's relative to the center cell, $TCM_{i,j}$, are also illustrated. DCM information may be maintained in independent, two-dimensional arrays or more complex data-structures. For example, the graph G might be maintained as a record containing the vertex value, eight DCM values, and minimum-cost direction pointer. Section 5.6 discusses the implementation considerations in choosing an appropriate data structure. Next, we consider the operations that must be performed on the data.

---

*As will be shown later, the path enumeration function uses less than 1% of overall processing time and the increase in storage required for the direction pointer may not be justified, unless it comes for *free*.

**The priority queue sorting algorithm makes a tremendous difference in the results. A linear sort method caused the algorithm to run several orders of magnitude slower than when using heapsort.

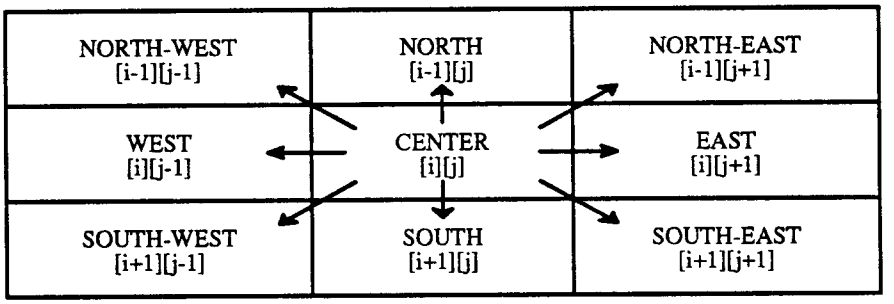


Figure 5-7. Rectilinear Array-structured Graph Minimization Operations.

The value at the center vertex is *minimized* with respect to each of the neighbor vertices and the cost associated with traveling to each of the neighbor vertices. If the cost *at* a neighbor plus the cost to travel *to* the neighbor is less than the center vertex value, the center vertex value is replaced with that cost. The neighbors are identified with the points of a compass, and the indices relative to the center cell at (i,j) are shown. Each direction has an associated DCM of the same dimensionality as the TCM.

5.4.4. Operations

By adding the cost at a neighbor TCM cell plus the cost to travel to that cell, and replacing the current center cell value with the calculated cost if it is less, minimum cost values are propagated to other TCM cells. An alternate, equivalent way to consider the operation is to replace the cost value of a neighbor cell if the sum of the current center cell cost plus the cost to travel from the neighbor cell to the current center cell is less. Then, repeat for all neighbors. Figure 5-8 suggests the general form of the replacement algorithm used to minimize each TCM cell for scan-based techniques. For Dijkstra's Algorithm, the current center cell is the one most recently added to the solution set.

$$TCM_{i,j} = min\Big[TCM_{i,j}, (TCM_{i-1,j} + DCM(N)_{i,j}), (TCM_{i-1,j+1} + DCM(NE)_{i,j}),$$
$$(TCM_{i,j+1} + DCM(E)_{i,j}), (TCM_{i+1,j+1} + DCM(SE)_{i,j}), (TCM_{i+1,j} + DCM(S)_{i,j}),$$
$$(TCM_{i+1,j-1} + DCM(SW)_{i,j}), (TCM_{i,j-1} + DCM(W)_{i,j}), (TCM_{i-1,j-1} + DCM(NW)_{i,j})\Big]$$

where,

$TCM_{i,j}$ is the current TCM cell,

$TCM_{i\pm1,j\pm1}$ are the eight TCM neighbor cells, and

$DCM(..)_{i,j}$ are the DCM values associated with the current center cell.

Figure 5-8. TCM Cell Minimization Operations.

To minimize $TCM_{i,j}$, near-neighbor TCM values must be accessed along with, $TCM_{i,j}$. Also, the corresponding eight DCM values at index (i,j) must be loaded. Then, the DCM values are added to near-neighbor TCM values and compared to $TCM_{i,j}$. Finally, the minimum value must be stored back in $TCM_{i,j}$, which completes the test, for a total of 34 fundamental operations per generalized cell check.

For any special purpose hardware implementation of path optimization, the ability to read and write the data efficiently during the sweeping and scanning processes is of primary importance. This may dictate how the data structures are represented in memory. Devices which can control their own addressing, coupled with multiported or interleaved memories can be used to realize parallel and/or pipelined architectures.

Although it is not part of the path optimization process, once the TCM has converged, finding the shortest path is simple: beginning with the starting point, simply pick the nearest-neighbor (NNR) cell whose TCM value plus the cost to travel from the current TCM is identical to the current TCM value. This is repeated recursively until the goal point is reached. The list of cells visited along the way becomes the path. It is not correct to simply pick the NNR TCM value that is the smallest, because the value at that cell plus the cost to travel there may exceed the TCM value there. Thus, it would *not* be one of the cells included in the minimum-cost path. However, there can be more than one path with the same shortest length; in this case, an arbitrary choice can be made.

5.5. Simulating the Algorithms

To determine the effectiveness of various algorithmic and scanning techniques, two software simulators were developed: and one for Dijkstra's Algorithm and one for scan-based algorithms.* The simulators allow one to specify arbitrary start and goal points, the array size, the number of no-change scans for scan-based convergence, the DCM values (either read or computed), the range of DCM values (i.e., minimum and maximum), the set of NNR cells to be used in the computation, the algorithm and/or scanning technique, and various debugging and map printing facilities.

Four data sets serve as input to the various algorithms under test: random integer values for DCM data, DCM values derived from elevation data corresponding to six cycles of sin(x)/(x) data, and two quartiles of the Digital Elevation Model of a one arc-degree section of the earth's surface centered on San Francisco (obtained from the United States Geological Survey [McEw85]). These are illustrated in Figure 5-9.

For random data, ten different sets of DCM values were calculated and used. Two different goal points (near the center of the array and near the upper left-hand corner) were chosen in order to observe any goal point dependence for each algorithm. These provide both boundary and nominal situations. All simulations used the same seed value with the random number generator, enabling a direct one-for-one comparison of the results.

For the Bay Area sections and sin(x)/x data sets, ten goal points for each data set were selected (shown in Figure 5-9). To produce DCM values, we use a relatively simple function of one variable, elevation. As mentioned earlier, the task of creating DCM values for real applications, such as land exploration, from a diverse collection of parameters is very involved. Useful results are obtained only when a meaningful function is used to generate DCM values. (For example, a simple linear mapping of the range of elevation differences to the range of possible DCM values yields questionable results — all paths are of equal costs, whether going up-hill or down-hill, and the calculated shortest path may lead straight over the highest peak!) A more detailed examination of the effects the DCM cost computation has on the various algorithms is important but beyond the scope of this dissertation, and will not be considered further.

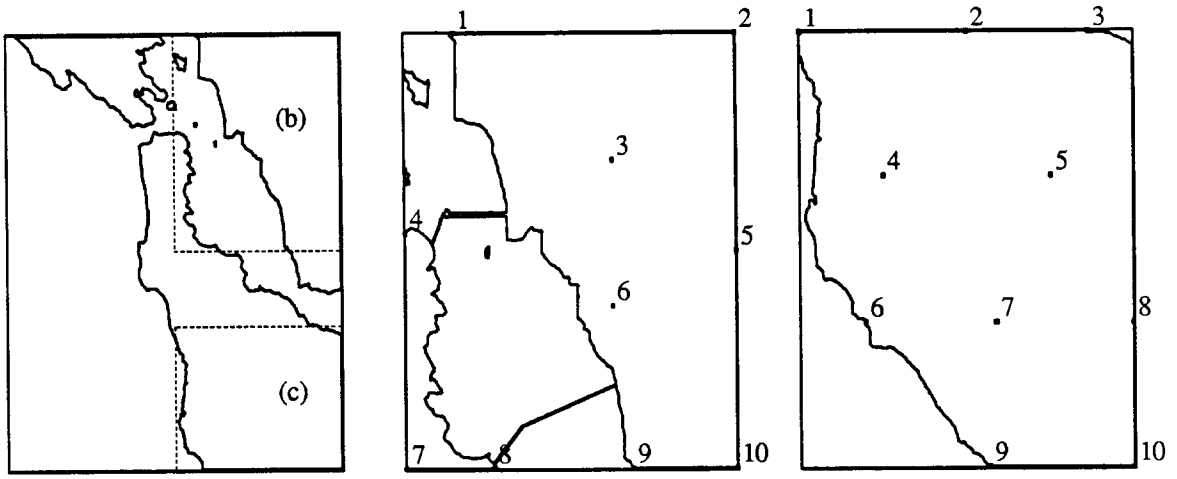
For our purposes, the DCM values are calculated in two steps. First, the differences between elevations at opposite ends of each arc are computed. Then, all values are scaled to fit between the bounds [1 .. 255] with the procedure shown in Equation (5-2) below.

$$DCM_{ij}(k) = ((E_{ij} - NNR(k)) + |DIFF_{\min}|)^3 * \frac{255}{(|DIFF_{\min}| + |DIFF_{\max}|)^3} \qquad (5\text{-}2)$$

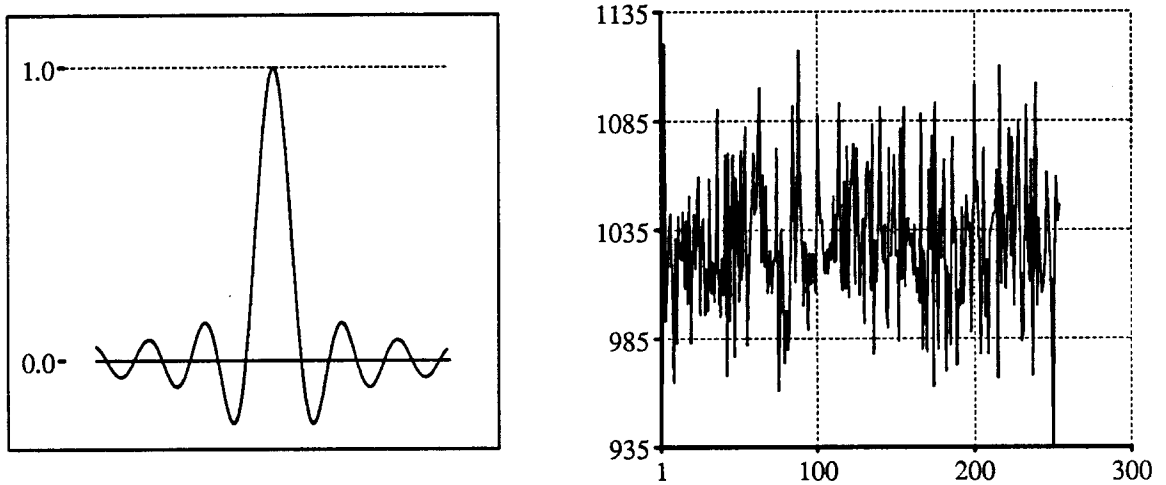
where

E_k = elevation of near-neighbor vertex in the k^{th} direction,
$DIFF_{\min}$ = smallest difference between NNR vertices, and
$DIFF_{\max}$ = largest difference between NNR vertices.

*The C-language code for both simulators is not included in this dissertation, but is available from the author upon request.



(a) San Francisco Bay Area (b) North Bay (c) South Bay



(d) Profile of Sin(x)/x (e) Histogram of 8-bit Random Numbers

Figure 5-9. Data Sets used for Shortest Path Simulations.

Parts (a), (b), and (c) show the San Francisco Bay Area, with (b) and (c) being the upper-right corner and lower-right corner of part (a) respectively. These data were obtained from the U.S. Geological Survey [McEw85]. The numbers (1 through 10) on (b) and (c) are the goal points used during simulation. Part (d) shows the elevation points (z-axis) along the x-axis (y=0) for a 3-dimensional surface created using a sin(x)/(x) generating function. Part (e) shows the histogram distribution of 256K data values in the range [1 .. 256] computed from random numbers generated by the UNIX RANDOM(3) routine. These values were used as elevation data to characterize a 512-by-512 point rectangular area. As expected, the mean number of points per value is roughly 1000. All DCM data that is either created or derived from the data sets is scaled to values between 1 and 255. The representation for the Oakland Bay and San Mateo bridges were added to the USGS data.

For each scan-based algorithm, measurements were made on convergence rates for the various sweeping techniques and TCM sizes. The convergence rate is determined by counting the total number of cell checks (i.e., test a cell against all its neighbors) until no change in the total cost map is produced for one macro sweep defined for the particular algorithm. This provides a basis for comparison between algorithms, assuming the other house-keeping aspects of the algorithms are insignificant. (Profiling the execution of the simulations revealed that usually less than 3% of the total execution time was spent on

routines other than *minimize_cell*). For scan-based techniques, the tests were run on array sizes of 8-by-8 up to 512-by-512 elements.*

5.5.1. Hypotheses Before Simulation

Before we started simulating the various techniques for path optimization, we identified four hypotheses we wanted to verify:

- Since results are data dependent, real data (e.g., from actual terrain maps) may converge faster for scan algorithms than random data due to the potential cell-to-cell *coherence* or smoothness.
- If typical terrain data does have some smoothness, perhaps a simplied near-neighbor test may be sufficient to determine minimum cost paths (i.e., just check N, S, E, and W neighbors and ignore those on the diagonal).
- Intelligent algorithms may show a marked advantage over brute force algorithms if the house-keeping functions necessary are kept to a minimum or are relatively simple. (If it is possible to add a node to the solution set with a small number of cell checks — on average two or four — and even if sorting structures are needed to maintain lists of neighbors, an overall speed advantage may result).
- The operand size may in practice be bounded, allowing a minimum number of bits for representing DCM and TCM values, resulting in a smaller amount of hardware.

The following section discusses the results of our experiments and focus on these four hypotheses.

5.5.2. Simulation Results

The simulation performance of Dijkstra's Algorithm for N = 512 is summarized in Table 5-2. The average number of cell operations is identical for all data sets and the average number of heap operations is consistent over all data sets. Execution time is data dependent but varies only a small amount between the data sets.

Table 5-2. Simulation Performance for Dijkstra's Algorithm, N = 512

Data Set	Ave CellOps	Ave HeapOps	std. dev	Ave HeapSize	std. dev
North SF Bay	259,335	5,041,087	7.1 %	1,128	31.6 %
South SF Bay		5,264,182	2.2 %	1,283	16.4 %
Sin(x)/x		5,378,015	2.9 %	1,375	25.7 %
Random		5,784,096	0.2 %	1,794	2.0 %

This table summarizes the performance characteristics of Dijkstra's Algorithm for TCM size 512-by-512. Each time a TCM cell is added to the solution set, the current-cost of the neighbor cells that are not in the solution must be updated, based on the current-cost of the most recently added cell. This is called a *cellOp*. On average, about four cellOps occur for each solution set entry. To make this comparable to scan-based techniques, four cellOps will be *equivalent* to one cellCheck (see the caption for Table 5-3). The heapOps are the number of comparisons made in establishing and maintaining an ordered heap (allowing a HeapSort algorithm for reordering after node deletion or addition). The heapSize indicates the maximum number of nodes in the heap at any time during the course of determining the total solution set.

The simulation results of the best overall scan-based algorithm for goal points near the border is Algorithm No. 7, summarized in Table 5-3. Algorithm No. 8 is slightly superior for goals more

*Although 512-by-512 may seem small, path planning systems typically change the resolution of the data sets, rather than scale to larger arrays. For instance, gross planning may provide data with 100 meters between points. Finer resolution planning may be 10 meters between points. In both cases, a 512-by-512 grid is used.

centrally located. The values shown are the average of 10 runs. For scan-based algorithms, the coefficient of variation of the number of cell checks versus the accumulated mean number of cell checks is less than 1% for seven maps or more. To illustrate this, Figure 5-10 shows two curves. The curve for random shows the number of cell checks needed for convergence of 10 different data sets. The curve for typical terrain data shows the number of cell checks for each of 10 different goal points for the North Bay data set. The number of operations needed to compute the TCM is constant for a given data set and algorithm, is independent of the host computer running the simulation (the amount of simulation time is all that changes), is a statistically well behaved function, and will serve as a reliable metric. Note that random data requires between 15 and 25 times more cell checks for convergence than terrain data.

Table 5-3. Simulation Performance for Scan-based Algorithm No. 7, N = 512

Data Set	Ave CellCks	std. dev	Ave Exch	std. dev	Ave Sweeps	std. dev
North SF Bay	2731050	12.1 %	253790	25.9 %	10.5	12.1 %
South SF Bay	1612620	21.2 %	291078	13.4 %	6.2	21.2 %
Sin(x)/x	2158830	47.5 %	359087	14.3 %	8.3	47.6 %
Random	42136200	3.5 %	7621530	4.6 %	162.0	3.5 %

This table summarizes the performance characteristics of scan-based algorithm No. 7 for TCM size 512-by-512. Each time a TCM cell is tested against its neighbors (an 8-way test) one cellCk is performed. Eight comparison/update operations occur for each cellCk. The number of comparisons resulting in updates and the number of sweeps to reach convergence are indicated. Note the large difference between random and typical terrain data.

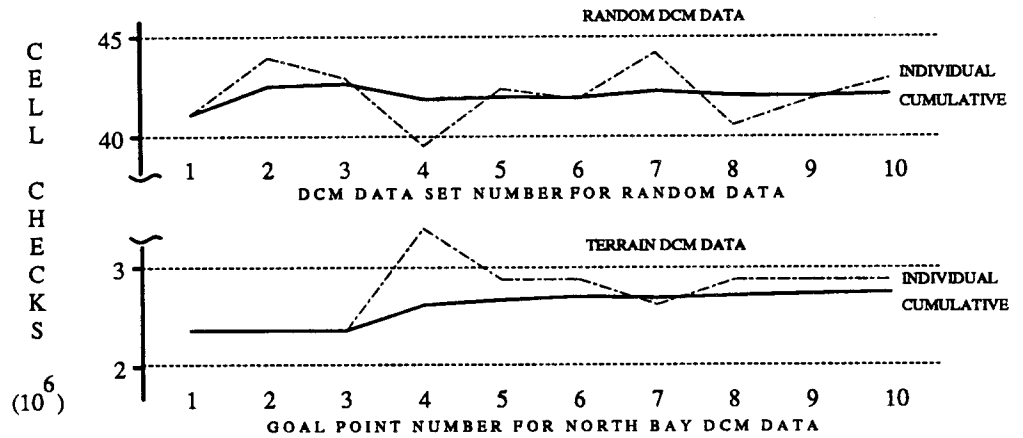


Figure 5-10. Number of Cell Checks for Convergence.

This figure shows the number of cell checks needed for convergence for 10 different random data sets and 10 different goal points in the North Bay terrain data set. The label for the X axis of the figure simply identifies the first, second, third, ..., through tenth random data sets and the first, second, third, ... through tenth goal point for the North Bay terrain data set. The dashed line represents the number of cell checks needed for each of the 10 individual data sets or goal points. The solid line is the *cumulative average* number of cell checks needed. This shows that the variation in number of cell checks to reach convergence for different data sets or goal points is very small. Also, the cumulative mean of the number of cell checks over 10 different data sets or goal points is a reliable and well behaved metric that will serve in making a comparison between algorithms. We do this to eliminate the bias that can arise from using a single data set or small number of data sets for our simulations. The other terrain data sets showed similar results.

Appendix C shows the results of simulation runs for all scan-based techniques. Table 5-4 summarizes results for both Dijkstra's and scan-based algorithms (goal-point numbers refer to Figure 5-9).

Table 5-4. Simulation Performance for All Algorithms (N = 512)
(selected goal points refer to Figure 5-9)

Data Set & Goal Co-ords	Ratio of CPU Usertime for Each Scan-based Algorithm vs. Dijkstra's Algorithm									
	Dijk (sec)	#1	#2	#3	#4	#5	#6	#7	#8	#9
North SF Bay #3 [150,320]	1.0 (36.6)	63.3	36.7	18.0	17.9	16.9	12.3	3.1	3.1	48.6
South SF Bay #7 [341,300]	1.0 (36.8)	109.3	32.8	15.4	14.6	16.7	11.1	2.2	1.8	1.6
Sin(x)/x #6 [085,341]	1.0 (36.2)	72.4	43.3	12.7	22.0	11.1	6.4	2.2	1.8	3.8
Random #3 [001,003]	1.0 (39.2)	133.7	66.8	126.0	75.6	93.2	86.2	54.8	59.8	181.9

This table summarizes the performance characteristics of all algorithms for TCM size 512-by-512 and various goal points (where the upper left-hand corner of the TCM is [0,0], the lower right-hand corner is [511,511]). These represent the ratio of execution time for each algorithm normalized to Dijkstra's Algorithm for the particular data set. (Similar results are found when comparing the number of cellChecks, a CPU independent factor.) For typical terrain data, Dijkstra's Algorithm is a factor of two to 19 times faster than scan-based algorithms. For random data, Dijkstra's Algorithm is between 55 and 182 times faster than scan-based algorithms. The cell-to-cell coherence of typical terrain data significantly affects the performance of scan-based algorithms, while the performance of Dijkstra's Algorithm increases only about 10% for random data over typical terrain data.

In comparing the run-time performance, we determined that Dijkstra's Algorithms is 55 times faster than the best scan-based algorithm for random data using a VAX 8850. Besides being significantly faster than scan-based algorithms, Dijkstra's Algorithm is essentially deterministic in the number of operations performed to achieve convergence. More work is done by Dijkstra's Algorithm for each TCM cell checked (including the maintenance of the priority queue) than with scan-based algorithms. But, Dijkstra's Algorithm allows the solution set to be constructed item-by-item as the algorithm progresses. This means that any given node is visited fewer times during the process, with commensurate savings in processing time.

As seen in Table 5-2, there is a small amount of variation in the number of operations needed to maintain a priority queue (heapsort algorithm) depending on the DCM data variability. The running time reflects this variation of about 13%, which is insignificant in comparison to the variability of the scan-based algorithms, as seen particularly between random and typical terrain data sets. Next, we consider the four hypotheses discussed in Section 5.5.1.

5.5.2.1. Random Data versus Terrain Data

Initially, all our experiments were carried out using random data. From those results, we determined that even parallel/pipelined architectures would have trouble achieving real-time performance with scan-based algorithms. However, the results of the experiment with typical terrain data confirmed our hypothesis that real data has cell-to-cell coherence, and all scan-based algorithms performed significantly better with real data, as seen in Table 5-5.

Before discussing data coherence and simplified comparison tests based on coherence, we briefly consider the performance of each of the algorithms, comparing what we anticipated when inventing them with the simulation results.

As shown in Figure 5-6, Algorithm No. 1 is the simplest scan-based method, and as we expected, this naive approach yielded the worst results. In one case, it was actually worse for terrain data than for random data — something no other algorithm can claim. We expected Algorithm No. 2 to be a slight improvement over Algorithm No. 1, and indeed it was, resulting in roughly twice the performance.

Table 5-5. Scan-based Algorithm Performance for Random Data versus Terrain Data (N = 512)

Data Set & Goal Co-ords	Number of Sweeps for Convergence with each Scan-based Algorithm								
	#1	#2	#3	#4	#5	#6	#7	#8	#9
North SF Bay #3 [150,320]	200	116	28	57	50	38	8	8	129
South SF Bay #7 [341,300]	340	102	24	46	52	35	5	4	3
Sin(x)/x #6 [085,341]	225	133	19	71	34	19	5	4	9
Random #3 [001,003]	223	206	202	256	299	272	161	190	264
Ratio of Number of Sweeps to Converge — Random Data vs. Terrain Data									
Random vs. North SF Bay	1.1	1.8	7.2	4.5	6.0	7.2	20.1	23.8	2.0
Random vs. South SF Bay	0.7	2.0	8.4	5.6	5.8	7.8	32.2	47.5	88.0
Random vs. Sin(x)/x	1.0	1.5	10.6	3.6	8.8	14.3	32.2	47.5	29.3

This table shows how scan-based algorithms compare for random and terrain data sets. The number of sweeps to converge is generally much larger for random data than the terrain data for any scan-based algorithm. The ratio ranges from close to 1.0 with Algorithm No. 1 to nearly two orders of magnitude with Algorithm No. 9. Note that Algorithm No. 1 is consistently poor in performance over all data sets, while No. 9 is superior for some and poor for others depending on the goal point.

Algorithm No. 3 appears to be a substantial improvement over either Algorithm No. 1 or 2 in terms of sweeps to converge. By reversing direction on each row, minimum cost information is propagated quicker, as we expected, but there is also twice as much work for each scan than with Algorithm No. 2. Thus, the overall improvement is only about a factor of two better than Algorithm No. 2. Interestingly, Algorithm No. 3 worked extremely well with the Sin(x)/x data and not well at all with random data.

We expected Algorithm Nos. 4, 5, and 6 to be improvements on the first three algorithms because they varied the starting point. While Algorithm Nos. 4 and 5 achieve about the same results, Algorithm No. 6, by flipping the starting point to all four corners of the grid, improved performance between 25% and 50% over the other two algorithms. This corroborated our intuition that pushing or pulling minimum cost information in all directions improves performance. Indeed, as shown with Algorithm Nos. 7 and 8, not only does starting the process in different corners help, but scanning in the top-bottom directions improved performance by a factor of between four and seven over other scan-based algorithms for terrain data. The improvement for random data is not as dramatic, but nevertheless substantial. Finally, Algorithm No. 9, which we expected to be the best, did achieve some good results. However, it tends to be less consistent over the different data sets and had a wide variability, which we did not expect.

5.5.2.2. Terrain Data Coherence and Simplified Neighbor Tests

Having established that terrain data converges much faster than random data, we expected that the cell-to-cell coherence of terrain data might allow a simpler comparison test, including only horizontal and vertical neighbors (i.e., a 4-way test). This is not the case. As illustrated in Figure 5-11, with the same start and end points, the path chosen for the 8-way test versus the 4-way test is significantly different. Similar results occurred on all terrain data sets and various start and goal points.

It is nevertheless important to realize that the largest contribution to the longer path length for 4-way tests may be due to the Manhattan constraint. On a flat plane it would take twice as many steps to

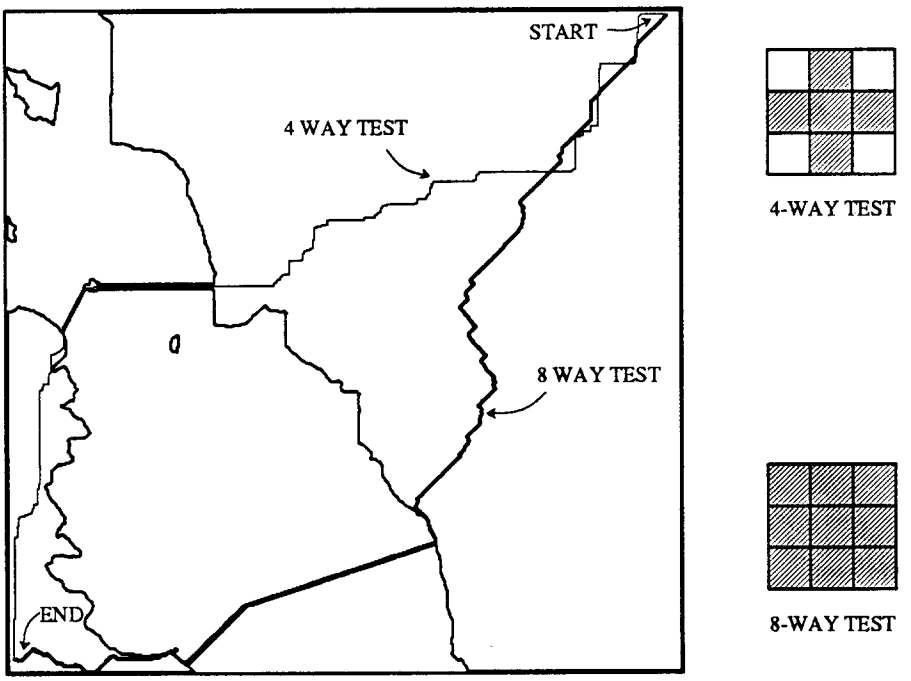


Figure 5-11. Shortest Paths for Four-way versus Eight-way Comparison.

For the starting point in the upper right corner of the North Bay data set, the shortest path to the lower left corner for 4-way NNR comparisons (thin solid line) is completely different from the 8-way comparison (thick solid line) result. The 4-way test to compute shortest paths yields non-optimal results. The resulting cost is data dependent, but was consistently observed to be between two and four times the 8-way cost.

move from one point to another point that is diagonally offset from the first, if arcs between diagonal points are weighted the same perpendicular points. Thus, the difference between 4-way and 8-way tests may not be as telling as the data shown in Figure 5-11 might suggest. On the other hand, the DCM data used here are based solely on elevation. If the actual distance between points enters into the DCM values, then a counter balancing effect comes from the fact that the euclidean distance between diagonal points is the square root of two times the distance between either of the perpendicular neighbors, and the factor of two reduction in the number of steps achieved for the 8-way test would reduce the path cost by a 30% instead of 50%. A more detailed investigation using a more realistic generating function for the DCM data sets is required to make a final determination, which goes beyond the scope of this dissertation.

To make a relative comparison of the different scan-based algorithms, we normalize the number of cell checks for each algorithm by Algorithm No. 8. Table 5-6 summarizes these results and Figure 5-12 illustrates the differences. Ratios greater than 1.0 represent algorithm-size combinations that perform worse than Algorithm No. 8.

From Table 5-6 and corresponding Figure 5-12(a), it is clear that for a goal point at an extreme corner of the TCM, Algorithm No. 2, No. 4, No. 6, No. 7, and No. 8 are essentially equivalent for all map sizes and 4-way tests. Moving the goal point to the center worsens the performance for Algorithm No. 2, No. 4, and No. 7 when N is increased. Nevertheless, Algorithm No. 3 is by far the best and Algorithm Nos. 5 and 9 are the generally the worst for 4-way tests.

The 8-way tests summarized in Table 5-7 and illustrated in Figure 5-12 (b) and for the goal point at TCM[1,3] differ considerably with 4-way test results. Here, Algorithm No. 7 is the best with Algorithm No. 8, Algorithm No. 2, and Algorithm No. 9 about equal. Notice that Algorithm No. 3, which was the best with 4-way tests is now the worst for both [1,3] and [C,C] = [256,256] goal points. During reverse scan, Algorithm No. 3 makes far fewer useful updates when using an 8-way test. Consequently,

Table 5-6. Normalized Average Cell Checks
(versus Algorithm No. 8 for 4-way test)

Algorithm	Array Size N (for NxN TCM)													
	008		016		032		064		128		256		512	
	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]
No. 2	1.12	1.24	0.93	1.62	0.88	2.01	0.98	2.55	1.09	3.21	1.11	3.74	1.21	4.53
No. 3	0.74	0.93	0.51	0.78	0.28	0.64	0.30	0.49	0.35	0.41	0.33	0.38	0.38	0.38
No. 4	1.12	1.12	0.95	1.22	0.88	1.48	1.00	2.01	1.11	2.36	1.08	2.65	1.16	2.65
No. 5	0.85	0.73	0.95	0.95	1.30	1.53	1.60	2.05	2.06	2.90	2.24	3.52	2.33	4.09
No. 6	1.14	1.00	1.05	0.91	1.01	0.99	1.02	1.06	1.02	1.04	1.01	0.99	1.01	1.00
No. 7	0.98	1.12	0.89	1.38	0.90	1.49	1.01	1.93	1.08	2.34	1.09	2.76	1.18	3.23
No. 8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
No. 9	1.27	1.09	1.54	1.54	1.66	1.96	2.37	3.05	2.68	3.79	2.63	4.13	3.45	5.27

This table summarizes the average number of cell checks for each scan-based algorithm relative to Algorithm No. 8, using a 4-way comparison/update test. Numbers greater than 1.0 indicate where an algorithm does not perform as well as Algorithm No. 8 for a given TCM array size.

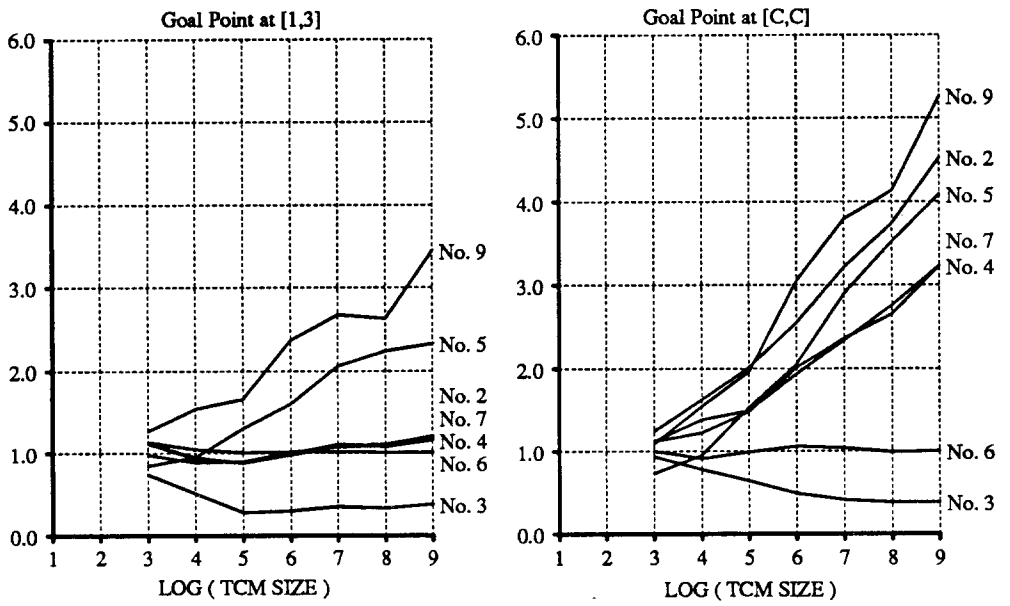
Table 5-7. Normalized Average Cell Checks
(versus Algorithm No. 8 for 8-way test)

Algorithm	Array Size N (for NxN TCM)													
	008		016		032		064		128		256		512	
	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]	[1,3]	[C,C]
No. 2	0.92	0.94	0.86	1.06	0.93	1.12	0.90	1.24	0.95	1.45	1.00	1.58	0.97	1.55
No. 3	1.36	1.57	1.38	1.51	1.70	1.52	1.71	1.59	1.79	1.74	1.94	1.84	1.89	1.69
No. 4	0.90	0.98	0.93	0.99	1.07	1.01	1.10	1.14	1.18	1.26	1.19	1.36	1.22	1.32
No. 5	0.80	0.90	0.99	0.99	1.26	1.01	1.28	1.09	1.34	1.23	1.36	1.30	1.35	1.26
No. 6	0.88	0.94	0.94	0.91	1.16	1.01	1.25	1.00	1.21	1.14	1.27	1.22	1.24	1.14
No. 7	0.94	1.02	0.93	1.06	0.93	1.06	0.87	1.07	0.84	1.19	0.82	1.30	0.77	1.28
No. 8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.10	1.00	1.10	1.00	1.10
No. 9	0.81	0.73	0.79	0.80	0.91	0.74	0.84	0.85	0.93	0.90	1.01	0.93	0.99	0.95

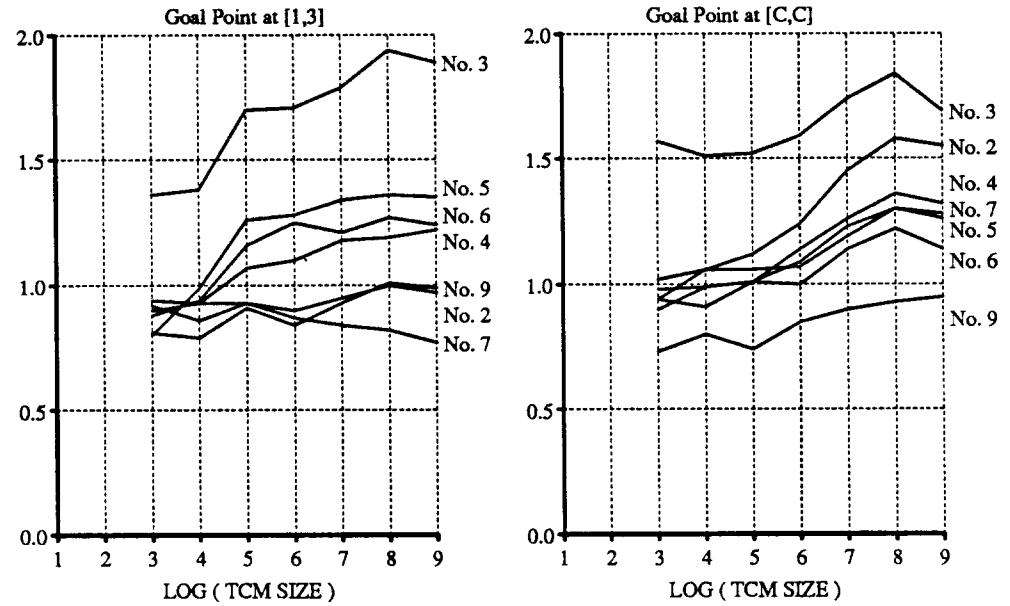
This table is identical to Table 5-6, except that an 8-way comparison/update test was performed with each cell check. Again, numbers greater than 1.0 indicate where an algorithm does not perform as well as Algorithm No. 8 for a given TCM array size.

about half of the scan cycles are essentially wasted. Algorithm No. 9 performs well in 8-way tests, but a special purpose hardware implementation would be difficult since the address sequence is goal dependent, not simply a raster-scan as with other techniques. However, with a general purpose host and coprocessor, the complexity of the addressing sequence is no more difficult than with the array scan techniques and offers some performance advantages.

As noted in Figure 5-12, 8-way tests are more consistent in performance across all algorithms. There are two reasons for this. With 4-way tests, some information (i.e, the diagonal neighbor TCM cells) is never considered when minimizing each TCM cell. Since there is a feed-back process involved (the dynamic programming aspect of the algorithms), fewer cells being considered per test limits the amount of feedback that is provided to minimize the current cell. Because of this, it takes longer for minimum cost information to propagate. Second, as we have determined, random data requires more comparison/updates before it converges. This accentuates the absence of feedback.



(a) Number of cell checks normalized to Algorithm No. 8 for 4-way comparison/exchange test.



(a) Number of cell checks normalized to Algorithm No. 8 for 8-way comparison/exchange test.

Figure 5-12. Normalized Average Cell Checks for Random DCM Data.

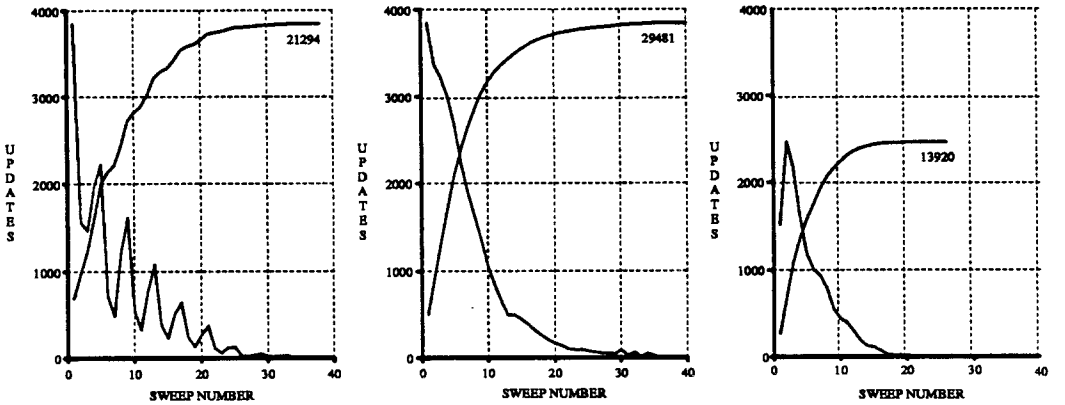
The average number of cell checks for 10 runs of each algorithm is normalized to Algorithm No. 8 for both 4-way and 8-way tests (see Table 5-6 and Table 5-7). For 4-way tests, there is nearly an order of magnitude difference between the best and worst algorithms in performance. For 8-way tests, the largest ratio is 2.5, showing a more consistent performance across all algorithms, as explained in the latter part of this section.

5.5.2.3. Intelligent versus Brute-force Algorithms

From Table 5-4, it is clear that Dijkstra's Algorithm is superior to all scan-based algorithms. The difference is less for terrain data than random data, as one would expect. If a special purpose architecture is to be implemented to achieve real-time performance, the advantage of the intelligent algorithm may be lessened by control complexity. There may be an advantage with a straight forward yet brute force algorithm due to its simple and regular form. In either case, kernel functions are the best candidates for hardware implementation, leaving the seldom-occurring control and processing functions to

software. These issues will be considered more fully in Section 5.6

To understand the differences between the sweeping techniques, the average number of updates versus the sweep number for each of the algorithms and array sizes is illustrated in Figure 5-13 and Figure 5-14. Superimposed on each plot is the cumulative number of updates. The dominant characteristics of updates versus sweep are evident with 64-by-64 TCM arrays. Larger array sizes look essentially the same and are not included here. Figure 5-13 shows the average updates per sweep for Algorithm Nos. 8 and No. 9 for 8-way tests and two goal points. A summary of the experimental results are included in Appendix C, Table C-1 through Table C-4.



(a) Algorithm No. 8, G[1,3] (b) Algorithm No. 9, G[1,3] (c) Algorithm No. 8, G[C,C]

Figure 5-13. Average Updates per Sweep for Algorithm Nos. 8 and 9.

Part (a) shows Algorithm No. 8 with an 8-way test and goal point at TCM[1,3]. The number of updates during each sweep depends on the direction of the sweep. Since the goal point was near the upper left corner of the array, sweeps moving from left to right and top to bottom will cause more updates than those moving right to left, or bottom to top, since the information can not propagate as far in those directions. Part (b) illustrates Algorithm No. 9 for the same dataset. For N = 64, Algorithm No. 9 requires about 15% fewer cell checks than Algorithm No. 8, but performs 38% more updates in the process, which can result in a substantial amount more memory traffic. The number of updates during each sweep is an exponentially decreasing function, since the goal point information is always pulled along with the cell checking process. In this case, it doesn't matter where the goal point is. The technique always propagates the goal point information *out and away* from the goal. Part (c) illustrates what happens to Algorithm No. 8 when the goal point changes to near the center. The first sweep does not result in nearly as many updates. Overall, far fewer sweeps and cellchecks are made to converge, with the accompaning reduction in the number of updates.

Algorithm No. 8 has four different sweep directions. Figure 5-13 (a) illustrates that relatively few updates are made during the second, third, and fourth sweeps of each four-sweep sequence. The number of updates is maximum when the goal point information is *pulled* along in the direction of the sweep. This occurs best when the goal point is near the beginning of the sweeping process, rather than at the end. The pulling process occurs even if the goal point is in the last cell checked. In this case, the minimum values are propagated one cell at a time each sweep, necessitating many sweeps to converge. Figure 5-13 (a) shows that for N = 64, Algorithm No. 8 requires between 30 and 40 sweeps to converge with random DCM data and results in over 21,000 updates. For Algorithm No. 9, the updates per sweep versus sweep number approximates a decaying exponential. This is expected, since the algorithm starts at an optimum point near the goal, and carries the minimum cost information along as it spirals around the goal point in an outward direction. Figure 5-14 illustrates the average updates per sweep for all algorithms.

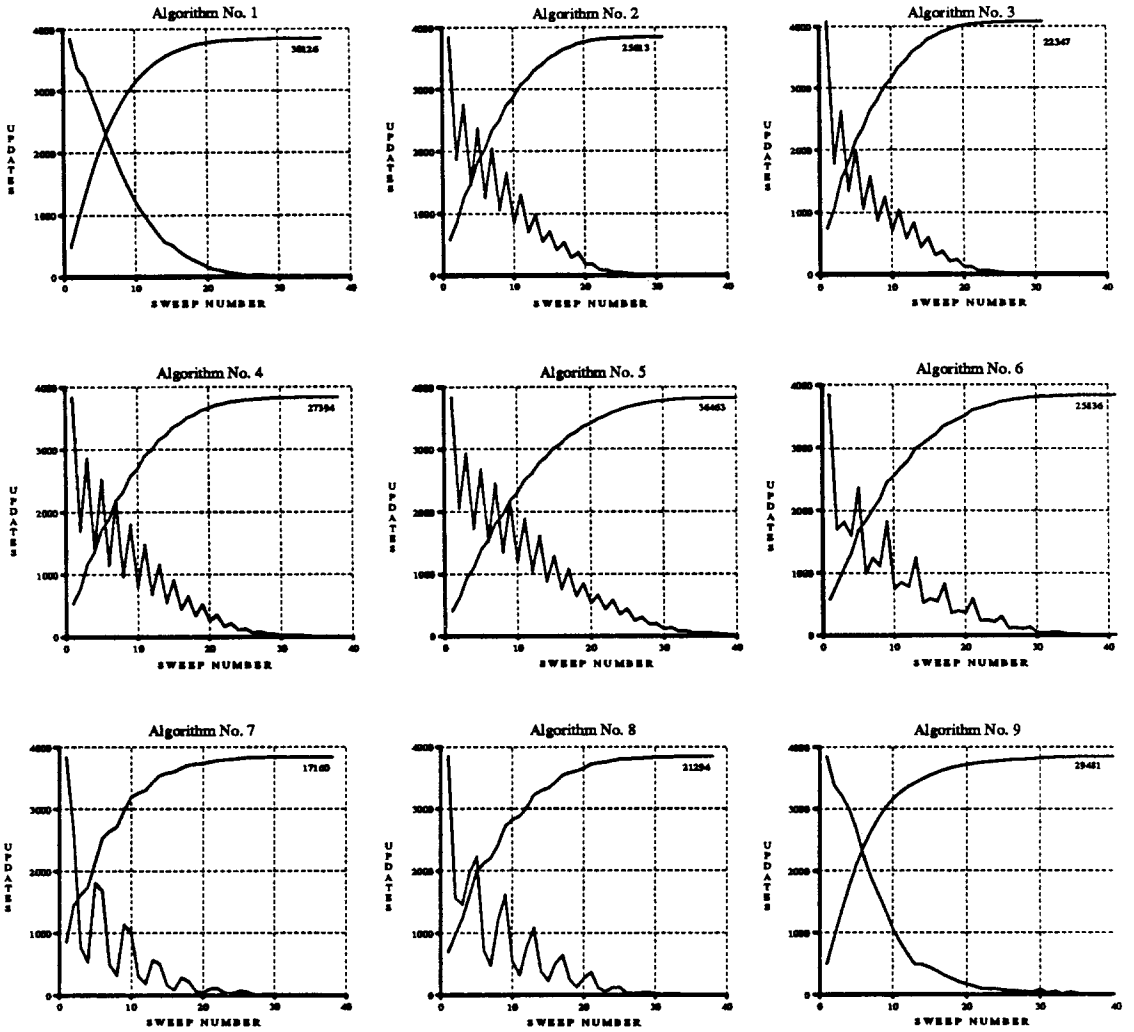


Figure 5-14. Average Updates per Sweep for All Algorithms.

This figure shows the average number of updates per sweep versus the sweep iteration number for Algorithm Nos. 1 through 9. The monotonic curve in each plot shows the accumulated number of updates to that sweep number, with the total number indicated near the end. The goal point was located at [001,003] for all simulations. As expected, the amount of useful work accomplished per sweep is a function of where the goal is located and the scanning/sweeping directions. For example, notice that Algorithm No. 4 makes very few updates on reverse-scan sweeps. Had the goal been located in the lower right-hand corner, the opposite would be true. Likewise, notice that Algorithm No 8 makes many more useful updates on sweeps one and four of each 4-sweep sequence than on sweeps two and three for the same reason.

5.5.2.4. Operand Size

The final TCM solution for a given goal point and given DCM's will be identical for all algorithms used. At any point in the TCM, the path cost to the goal is indicated by some data-dependent value. The maximum TCM value (the highest-cost path from a point in the TCM to the goal) depends on the data as well as the number of cells that represent the area of interest. As the size of the TCM increases, the minimum cost path from border cells to the goal point will, on average, inevitably be *longer* and possibly larger in value. Therefore, the quality of the solution can be adversely affected unless there is sufficient dynamic range in the representation of the TCM to accommodate large values.

If DCM values are sufficiently large, the value calculated for a TCM cell replacement could eventually overflow the number of bits used to represent a TCM value. This could result in regions of the

TCM with the same *apparent* value, and would make the path determination process difficult if not impossible. To guarantee that TCM values do not overflow, and are less than the initialization value, the maximum TCM value must be determined. But, the largest TCM value is also data dependent. In the worst case, the shortest path could progress in a serpentine fashion through the TCM including every cell to reach the goal.

To obtain an estimate of how large the TCM values may get, we did a simulation using random data for the DCM's with the range [1 .. 1023] (1 to 10 bits). For TCM array sizes up to 512-by-512 elements, the maximum TCM values are summarized in Table 5-8 and illustrated in Figure 5-15. These results suggest that if DCM data are limited to eight bits, 16 bits will be adequate for the TCM values (for a 512-by-512 TCM) providing roughly a factor of three margin. The arithmetic add in the minimization step should be implemented with saturating adder circuitry. If overflow is detected, rescaling of the input data would be necessary.

Table 5-8. Maximum TCM Values From Simulation Using Random DCM Data
Goal point at TCM[1,3]

Max DCM	TCM Array Size (for N-by-N)							DCM	TCM
Value	8	16	32	64	128	256	512	Bits	Bits
1	1	1	1	1	1	1	1	1	1
3	4	6	8	10	9	9	10	2	4
7	9	22	35	60	110	194	360	3	9
15	29	54	97	168	318	599	1154	4	11
31	59	104	206	380	760	1382	2720	5	12
63	105	238	430	834	1563	2940	5747	6	13
127	243	505	917	1665	3147	6256	12137	7	14
255	492	1090	1689	3544	6726	12719	24860	8	15
511	803	1469	2979	6057	12112	25403	49926	9	16
1023	1763	3395	6714	12903	24813	50996	101786	10	17

This table shows the maximum TCM values for different DCM sizes and TCM array sizes using random data. Ten maps were simulated for each TCM size and DCM bit-width combination. Each entry represents the largest TCM value for all 10 maps. Terrain data sets resulted in maximum TCM values at least 33% smaller than these numbers for all array DCM sizes. TCM values are typically seven bits wider than DCM values.

5.5.2.5. Algorithm Complexity

By forming the ratio of the number of cell checks to the array size (raised to various powers), Figure 5-16 suggests that for arrays larger than N = 64, all algorithms are essentially $O(n^{1.5})$, somewhat better than $O(n^2)$ as predicted in the discussion in Section 5.4.2. Since this is a ratio of cell checks to array size, the more cell checks done by a technique in solving a map, the less efficient it is unless pipelining techniques can be used. Algorithms with lower valued ratios should be selected for implementation. Nevertheless, a factor of four to six in theoretic running time may not be significant, and the algorithmic technique that is simplest to implement may prove to be the best overall. Dijkstra's Algorithm is slightly better than scan-based algorithms in complexity, approximating $O(n^{1.25})$.

The previous sections have considered software implementations of the algorithms. In the following sections, several alternatives to software are discussed.

5.6. Other Software and Hardware Implementations

The algorithm analyses were completed on conventional computers using the programs described earlier. The literature also reports a number of implementations that have considered other architectures, both conventional and special purpose. Table 5-9 summarizes some of the results reported here

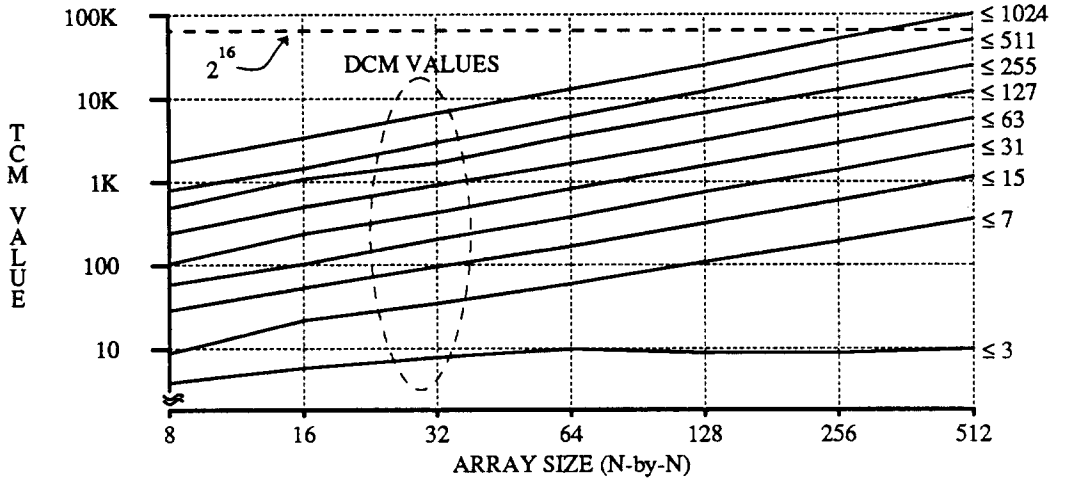


Figure 5-15. Maximum TCM Value for Various TCM Array Sizes and DCM Maxima.

This is plot of the data in Table 5-8. For DCM values greater than seven, TCM values increases for increasing TCM array size in a log-log correspondence. These results come from using random DCM data, and exceed the maxima resulting from terrain data TCM's, in all cases. The dashed line represents the maximum value representable by 16 bits.

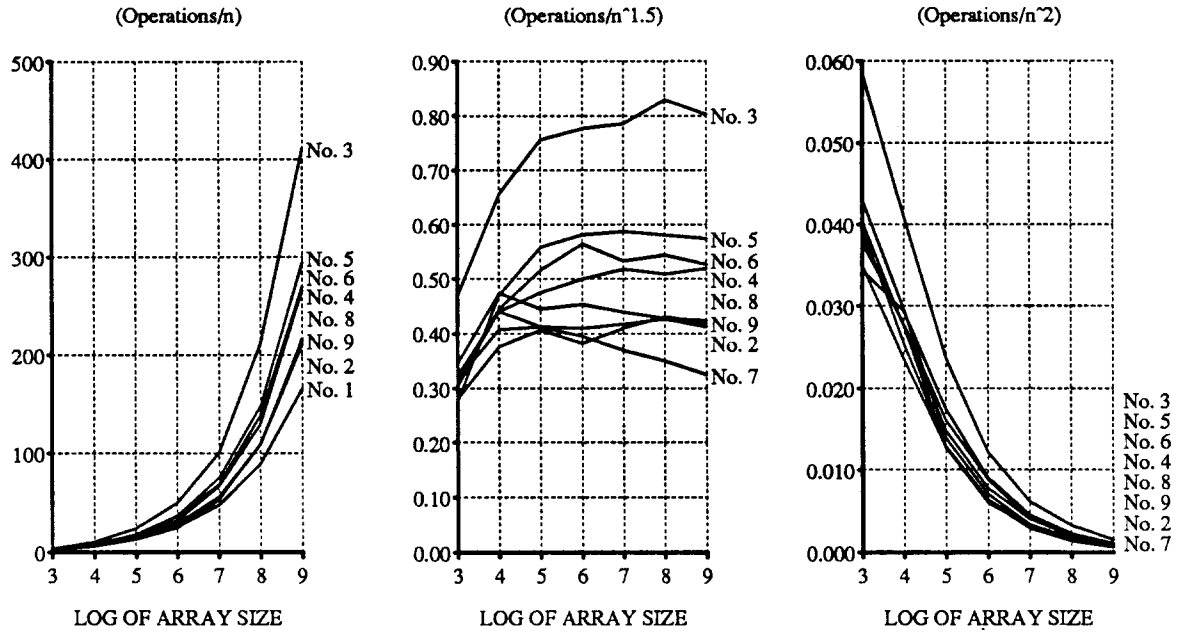


Figure 5-16. Scan-based Algorithm Complexity.

This figure illustrates the complexity of the various algorithms as determined by experiment. For array sizes of $N-by-N = n$ elements, the complexity for scan-based algorithms approximates $O(n^{1.5})$. Dijkstra's Algorithm is slightly better, with $O(n^{1.25})$.

and elsewhere.

We have identified and simulated various algorithms for path optimization considering performance, complexity, dynamic range of the data, and so forth. Next, architectures to implement these algorithms are considered.

Table 5-9. Cell Checks per CPU Second
(Normalized to VAX 11/780 Cell Checks per CPU Sec)

Manuf or System	CPU or Model	Clk Rate (MHz)	Mem Size (Mbytes)	8-Way Cell Check	
				Ck/Sec	Ratio to 780
Sun2	MC68010	10	4	2280	0.68
VAX	780	5	8	3340	1.00
BBN*	5 node B-fly	na	na	3930	1.18
VAX	785	7.5	12	5250	1.57
Pyramid*	98x	10	na	6680	2.00
Sun3	MC68020	16	8	6890	2.06
Sun3	MC68020	25	32	12500	3.74
VAX	8800	22	80	15850	4.75
BBN	40 node B-fly	10	na	31440	9.41
IBM	3090	54	38	59740	17.89
Par/Pipe HW**	Sp. Purpose	20	2.5	~20 M	~5800
Thinking Machines	CM2 (64K nodes)	6.7	64K bits per node	~150-400 M	<400000

This table shows the performance resulting for various software implementations and special purpose hardware implementations of scan-based algorithms (in most cases, similar to Algorithm No. 7). The single asterisk indicates extrapolated values, double asterisk indicates simulated values, and all other values come from measurements.

5.6.1. Multi-Processor Path Optimization Architectures

The path optimization function consists of data movement, integer addition, comparison, and replacement. For 512-by-512 arrays, the 34 operations necessary for each cell check (worst case for most scan-based algorithms), approximately nine million operations per sweep must be completed. Scan-based algorithms require about 200 sweeps to converge for random DCM data, or roughly 1.8 billion operations. The sequence of assembly language instructions needed to implement those 34 operations exceed 20 billion. At a 10 MHz clock rate, assuming a single cycle per instruction and/or operation, 200 sweeps would require about 2000 seconds, or three orders of magnitude slower than needed for real-time operation. Performance using terrain data is slightly more than an order of magnitude better. Nevertheless, the performance achieved for software implementations of scan-based algorithms is completely inadequate. In order to achieve real-time performance, some means of exploiting concurrency must be found.

Processor architectures are often classified according to how the control unit handles the sequence of instructions and/or the execution unit handles the data that is manipulated. Flynn in [Flyn72] discusses four categories of computer architecture based on this notion:

- SISD - single instruction, single data,
- SIMD - single instruction, multiple data,
- MISD - multiple instruction, multiple data, and
- MIMD - multiple instruction, multiple data.

Kuck [Kuck78] modifies this taxonomy somewhat by suggesting that it is easier to make the distinction between real machines based on the execution stream model (yielding SISE, SIME, MISE, and MIME categories), rather than the data stream model. He also considers vector operations versus scalar and how one might describe such architectures. Flynn's categorization is sufficient for our purposes.

Next, our architectural analysis for a path optimization processor considers a multi-processor evaluation and a detailed single coprocessor architectural study. The multi-processor evaluation considers the various types of parallelism that can be used to solve the problem. Significant performance

improvements can be obtained if cell operations can be done in parallel and multiple processors operate on a single TCM. Multiprocessor architectures and parallel/pipelined organizations are more likely to provide the needed performance but at increased cost.

5.6.1.1. SIMD

A SIMD approach would use one processor per TCM array cell. With this architecture and near-neighbor connections, one entire sweep could be completed in the amount of time needed to perform one cell check, or one *logical clock tick*. In other words, nine million operations per sweep would be reduced to 34 sequential operations at most. All processors perform the same operations simultaneously, but on different data. For such an architecture, Table 5-10 and Figure 5-17 show the performance characteristics of a functional implementation of such a connection-array *parallel-flash* algorithm, where each node has an associated (possibly virtual) processor.

It appears that more sweeps are required for the connection-array algorithm to reach convergence than with any other algorithm. The reason is that processors are always using *old* neighbor TCM values when making comparisons for updating a current TCM cell. With the other scan-based cell-by-cell algorithms, an updated TCM cell can enter into the calculation of the next TCM cell in sequence immediately after it has been calculated. With the connection-array parallel approach, all cells are updated simultaneously, and therefore will not influence the calculation of a neighbor cell until the *next sweep* which is the next functional clock tick. As a consequence, a uniprocessor (SISD architecture) emulating a parallel algorithm has much poorer performance than almost every other algorithm.

Table 5-10. Random versus Terrain Data Performance for Connection-array Algorithm
(Parallel-flash simulated for N=512)

Data Set and Goals Coords	Number of Sweeps for Convergence	
	Alg No. 8	Alg No. 12
North SF Bay, #3 [150,320]	8	685
South SF Bay, #7 [341,300]	4	543
Sin(x)/x, #6 [085,341]	4	890
Random, #3 [001,003]	190	1750
Ratio of Number of Sweeps to Converge — Random Data vs. Terrain Data		
Random vs. North SF Bay	23.8	2.6
Random vs. South SF Bay	47.5	3.2
Random vs. Sin(x)/x	47.5	2.0

This table summarizes the performance characteristics of the connection-array parallel-flash algorithm for TCM size 512-by-512 and various goal points. The algorithm assumes a SIMD parallel implementation. Although it appears that more sweeps are necessary for convergence than other algorithms, the amount of time per sweep is many orders of magnitude less, due to parallel operations.

Simulation results for a SIMD architecture indicate between 500 and 1800 logical clock ticks would be necessary to converge 512-point TCM. With a nominal 100 nanosecond cycle time, the 34 operations would take less than seven milliseconds, worst case. However, this would require 256K processors, and would not likely be cost effective, even if implemented on an architecture like the Connection Machine [Hill85] with virtual processors. Figure 5-18 illustrates the anticipated performance of SIMD architectures for different numbers of processors, up to a maximum of one processor per cell. Time-shared virtual processors are assumed when there are fewer processors than the number of TCM cells. The size of available local memory may limit the lower bound on the number of processors that can be effectively used.

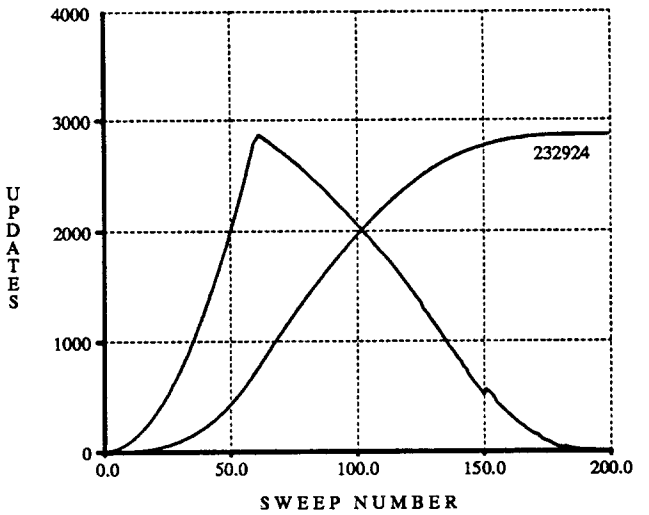


Figure 5-17. Average Updates per Sweep for Connection-array Algorithm.

This figure shows the average number of updates per sweep versus the sweep iteration number for the connection-array algorithm. The goal point is located at [001,003]. The characteristics of this algorithm are quite different from all scan-based algorithms. As illustrated, the number of updates starts off very small for the first sweeps, increases as the goal point information propagates, and then decreases during the final phases of the algorithm. For this algorithm, information from the goal point propagates only one cell per sweep. But, each sweep is many times faster than scanning techniques and is overall one of the fastest possible implementations.

5.6.1.2. MIMD

Approaches using commercially available multi-processors in a loosely-coupled architecture (on the order of 10-100 CPU's) achieve linear speedup simply by partitioning the TCM, as illustrated in the work by [Lind86]. The minimization of each cell requires access to its neighbors in the TCM. To avoid access conflict between independent processors, the TCM can be divided into regions or blocks, which can be assigned to each processor exclusively. The simplest way to partition the TCM is by rows. For a 512-by-512 array, two blocks of 256 rows by 512 columns would be needed to support two independent processors. For four processors, four blocks of 128 rows and 512 columns each would be formed. In general, the number of rows per block would be the array dimension divided by the number of independent processors. Each independent processor's cache would capture and hold its portion of the data set, making its effective access time essentially that of the cache memory. Each processor works on one portion of the TCM and boundary values are automatically communicated to other processors as needed through cache misses and a cache coherence mechanism. It is possible to loosely synchronize the processors so they will not reach their common borders simultaneously, and degrade performance by *ping-ponging* shared data values between their caches.

The overhead of the cache-miss communication does not become a significant factor as long as the number of processors is reasonably less than N, the size of the TCM. Although each processor is performing the same algorithm, they are not strictly synchronous. Such an architecture is termed MIMD. For a SPUR-based multiprocessor system, our simulations show that the hit ratio in the direct-mapped 128 Kbyte cache is greater than 99.5% for a uniprocessor running scan-based algorithms. That would change substantially as the number of processors increased and the data contention and communication time would become prohibitive. Figure 5-18 illustrates the theoretical performance for such an MIMD architecture.

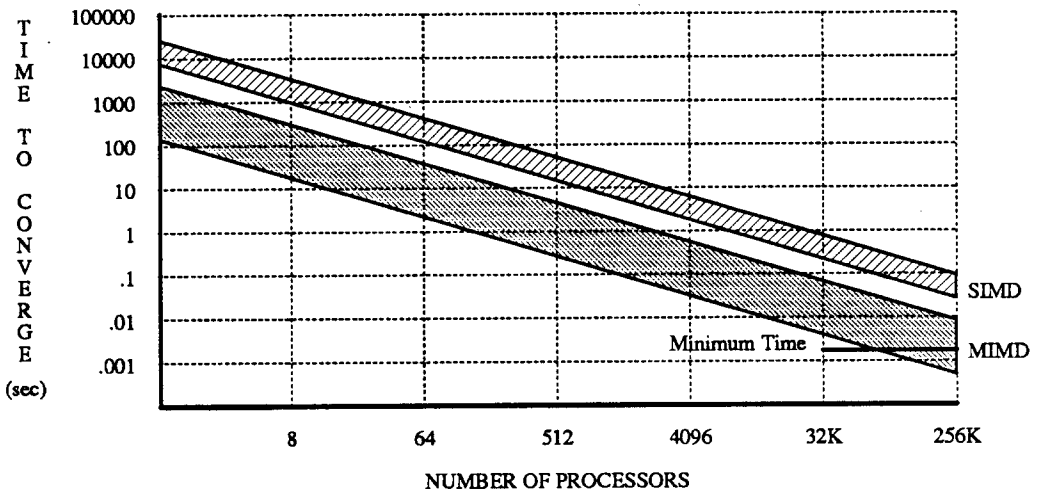


Figure 5-18. Convergence Performance for SIMD and MIMD Architectures.

This figure illustrates the performance of SIMD and MIMD architectures applied to path optimization. Performance is data dependent and the shaded regions indicate the amount of time to process a typical 512-by-512 data set (either terrain or random data) for different numbers of processors over a variety of data sets. For example, a 512-processor SIMD configuration would need between 13 and 50 seconds to converge. For a given number of processors, the MIMD architecture shows better theoretical performance than SIMD, but the overhead associated with communicating data with the larger number of processors would make it ineffective. It should be noted that the length of the worst case path would be about the same as the dimension of the TCM array. This sets an absolute minimum processing time of about 1.7 milliseconds as shown by the *minimum* line.

5.6.1.3. MISD

For tightly coupled multi-processors with a single shared memory and software pipelining, a specialized sweeping algorithm (e.g. the spiral approach) appears best. In this case, a processor pipeline would perform the operation, with processors staged one behind the other, each of which would be performing a sweep. To signal completion, two successive processors completing sweeps without change would cancel the remaining sweeps (still in progress). However, considering the amount of conflict at the memory, the effectiveness of the processor caches would be nullified. If the processors shared the cache, the problem of the memory latency time could be avoided for all the cache misses, but the contention at the cache (assuming single port) would make the process at least a factor of two slower than the MIMD architecture with partitioned TCM arrays. Thus, this architecture proves to be too cumbersome, and will not be consider further.

5.6.2. Hardware Architectures and Implementations

As an alternative to uniprocessor or multiprocessor configurations relying on standard software, special purpose coprocessors will be considered in this section.

5.6.2.1. Dedicated Special Purpose Devices

At the extreme end of the performance spectrum are hardware implementations that require little if any host interaction or support. These are essentially stand-alone devices capable of performing the entire algorithm in hardware. The advantage of such implementations comes from being able to tune the hardware to the exact needs of the algorithm and associated data structure. Optimum performance is achieved by providing special purpose dedicated memories to hold each of the DCM and TCM arrays, thus avoiding access conflict at the memory. The algorithm selected is based on the ease and regularity of implementation more than the outright best relative performance, as reported in Section 5.5.2 earlier. Regularity can be used to advantage in providing address generation for memory access. Parallel and pipeline structures that take full advantage of the order of operands flowing out of and into

memory can effectively provide one TCM cell update each clock cycle. In such cases, assuming the nominal 100 nanoseconds clocking rate, a single sweep of a 512-by-512 TCM would take only slightly more than 26 milliseconds. For terrain data and a good scan-based algorithm, 10 sweeps for convergence would amount to about 0.25 second, well within the goal of sub-second TCM convergence.

As our simulations have shown, scan-based techniques are data dependent. This imposes a disadvantage for architectures implementing such algorithms, because the convergence time is unpredictable. Nevertheless, it is constrained within certain bounds. In the case of completely random data, our studies show that the convergence rate for a dedicated parallel or pipelined architecture can be as much as five seconds. However, this is highly unlikely for typical terrain data, and would certainly represent a conservative upper bound. Other applications (maze routing, VLSI routing, and so forth) need further investigation to determine the processing time variability as a function of the input data and processing requirements. Figure 5-19 illustrates a parallel architecture similar to that reported in [Mars80], and Figure 5-20 illustrates a pipelined implementation similar to the one in [Hans87].

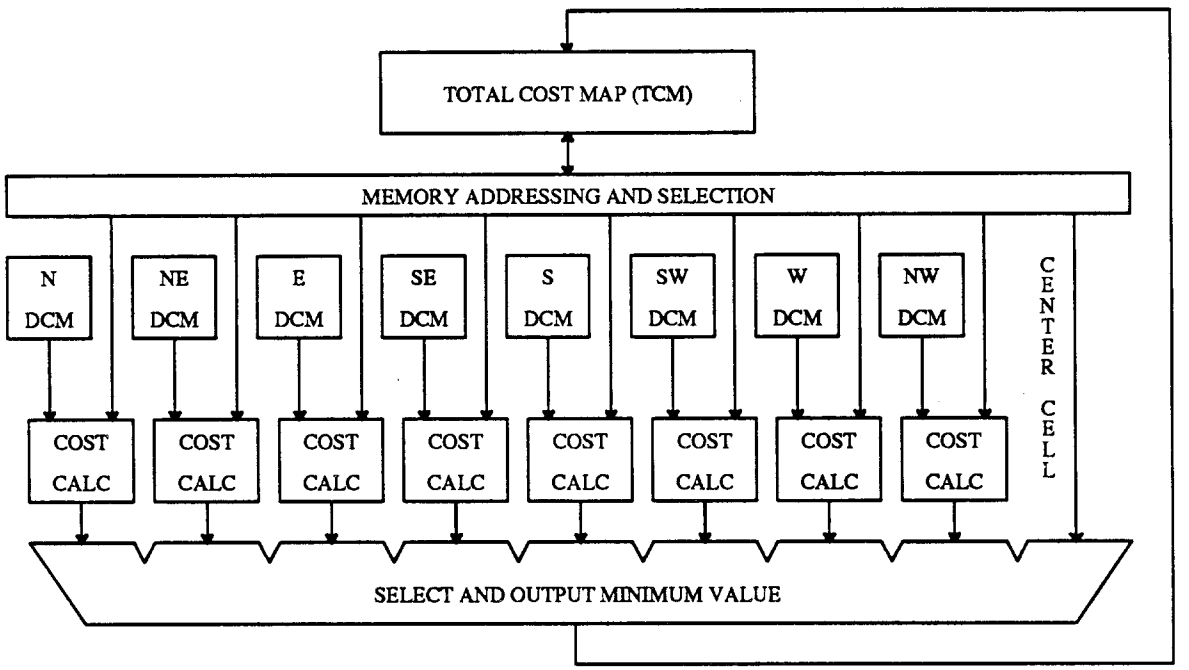


Figure 5-19. Parallel Dynamic Programming Architecture.

This figure illustrates the fundamental components in a parallel/pipelined architecture for dynamic programming applied to path optimization. The architecture requires a 9-way comparison, which is implemented in a comparison tree [Mars80]. Recently minimized cell values are used in computing current minimum costs.

5.6.2.2. Kernel Function Coprocessors

The speed advantage of dedicated hardware devices is often difficult to justify because of the relatively large expense and inflexible nature of such one-function implementations. The kernel functions of the path optimization process are addition and comparison. A path optimization coprocessor designed to implement these functions quickly is considered in the next sections. Many of the housekeeping operations, such as address generation, operand load/store, and so forth are left to the host CPU software.

5.6.2.2.1. Dijkstra-based Path Optimization Coprocessor

Dijkstra's Algorithm is far superior to scan-based algorithms for software implementations. The algorithm execution time is nearly constant, whether terrain data, pseudo-real sin(x)/x, or random data

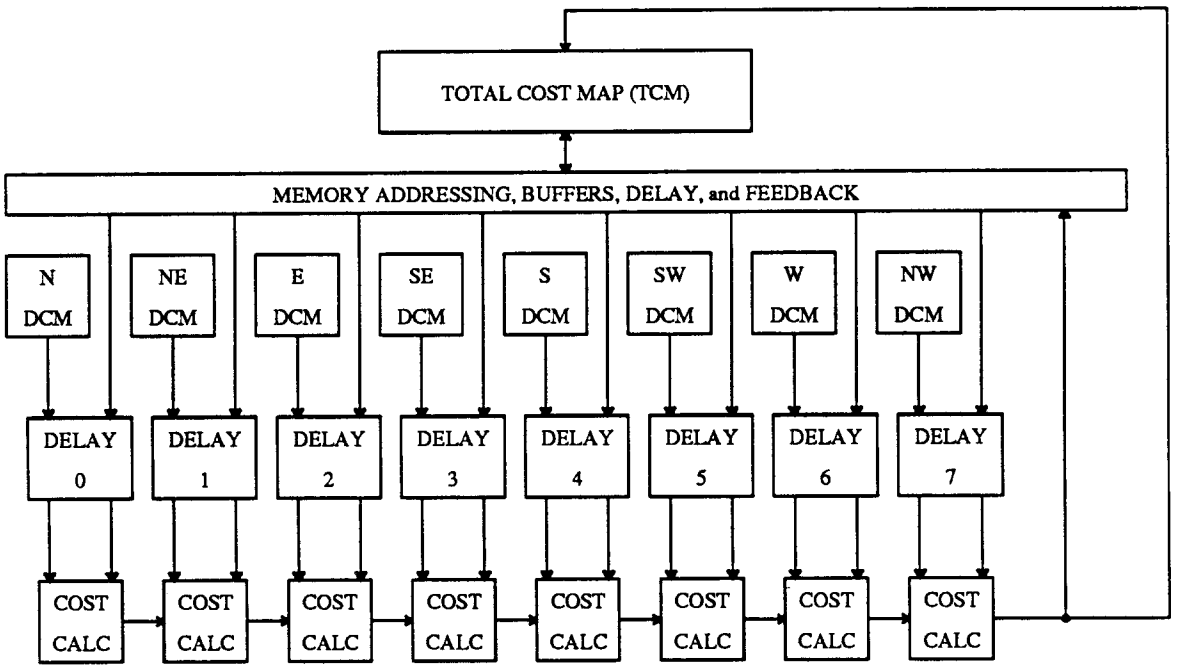


Figure 5-20. Pipelined Dynamic Programming Architecture.

This figure illustrates the fundamental components in a pipelined architecture for dynamic programming applied to path optimization. The architecture relies on operands being available in sequential pipelined fashion. This necessitates buffering several rows of data for use in comparison operations and for writing results back to the TCM [Hans87].

were used to generate the directional cost maps. By profiling the software execution, it was determined that 35% of the execution time was spent in the cost update function (e.g., *minimize_cell* of the scan-based algorithms), and 65% of the time was spent in maintaining a priority queue for the perimeter list. If the cost computation function became insignificant through the application of a coprocessor dedicated to that function, execution time would decrease by only 33%, or roughly 20 seconds of the one minute run-time on a VAX 8850. This applies to virtually all data sets.

Applying a coprocessor to the task of maintaining a priority queue has been discussed in previous work [Care85]. However, we determined that the most efficient implementation of Dijkstra's Algorithm requires that items in the queue be accessed randomly when applying heap-sort to maintain heap order of the priority queue. Most VLSI priority queues rely on a systolic architecture which allows access only at queue ends, rather than the needed random access. A content addressable portion of the sorting queue could be incorporated to solve that problem. With the current state-of-the-art in static rams, and allowing for a 16-bit key and 18-bit tag, approximately 1K entries could be fabricated on one chip. Dynamic memory technology would allow approximately an order of magnitude larger array to be built. Our simulations indicate that the maximum length for such a priority queue is less than 2500 elements. By providing a cascading feature as discussed in [Care85], three static priority queue sorter and content addressable memory chips could implement the function. Finally, to maintain proper sort order after changing an entry in the middle of the priority sort queue, a number of compare exchanges equal to the position of the element in the queue are necessary. Our simulations indicate that approximately 1024 would be needed.

We have also found that the priority queue is resorted approximately 500,000 times during a typical 512-by-512 problem. Thus, with a nominal cycle time of 100 nanoseconds, and presuming that integer and loop overhead instructions to be buried in the iterative compare updates, the time for solving the TCM for such an architecture would be approximately 50 seconds. Clearly, a priority queue that allows random access to its elements is essential for the efficient realization of this algorithm with a coprocessor. Although such a coprocessor is feasible, it would require either its own large, private

memory or direct access to the host's memory to make it effective, and even then it would be too slow. Such an architecture is more like the dedicated special purpose hardware devices described previously and will not be considered further. Nevertheless, when considering a software implementation, Dijkstra's Algorithm is superior to scan-based algorithms. The following section considers the architecture of a scan-based coprocessor.

5.6.2.2.2. Scan-based Path Optimization Coprocessor

To facilitate the operation of a coprocessor designed to execute scan-based algorithms, the necessary data structures must be reconsidered. For highly pipelined, dedicated architectures, independent memories and distinct arrays in contiguous sections of the memories were dedicated to TCM values and the eight sets of DCM values. Such memory configurations are necessary to allow these architectures to access memory simultaneously.

On the other hand, a coprocessor coupled with a general purpose CPU must rely on the behavior of the cache to provide rapid access to operand data. In such a case, a composite data structure produces better cache performance. If no special purpose memories are used, each node in the host processer memory consists of eight bytes of DCM information and two bytes of current minimum-cost value information for the cell. This amounts to 10 bytes of information for each cell requiring three accesses for a 32-bit architecture. A 512-by-512 problem would require 2.6 Mbytes of data space. For practical purposes, this becomes 3.0 Mbytes if 16-bit TCM values are used and the two extra bytes in the TCM word are either ignored or used for minimum_direction pointers. In the context of the SPUR CPU, 64-bit wide transfers are possible between the cache memory and the coprocessor. For addressing simplicity, the memory layout might consider 16 bytes per element, necessitating 4.0 Mbytes of data space. These and other possible layouts for TCM and DCM data in memory are illustrated in Figure 5-21.

Using the data structure shown in Figure 5-21b, the address of all data needed to minimize a particular center cell can be easily derived from the logical [row][col] address in the TCM matrix. Table 5-11 illustrates the address computation needed.

As outlined earlier, a cell check logically consists of nine loads of TCM data, eight loads of DCM data, eight additions of TCM(CC) and the DCM data, eight comparisons of newly computed interim center cell values, and one store of the result in the center cell. With the data structure and addressing scheme explained above, the center cell current minimum value is loaded in the coprocessor. Second, the DCM values associated with that cell are loaded. Then, as each of the neighbor cells current minimum information is loaded, the coprocessor performs the addition, comparison, and select minimum function in pipeline fashion, keeping pace with the load sequence after some delay. After the eight comparisons are complete, the result it written back to memory. Figure 5-22 (a) illustrates a straight forward implementation of a scan-based algorithm. With such a scheme, a cell check nominally consumes 14 cycles with a 32-bit host architecture providing single-cycle instruction times. With SPUR, this can be reduced by one cycle due to the 64-bit wide interface between the coprocessor and cache memory. At 100 nanoseconds per cycle, this translates into approximately 0.35 sec per sweep. For typical terrain data requiring between four and 10 sweeps to converge, this results in a convergence time of approximately 1.4 to 3.5 seconds, which does not meet the sub-second performance goal, but nevertheless is significantly better than software implementations.

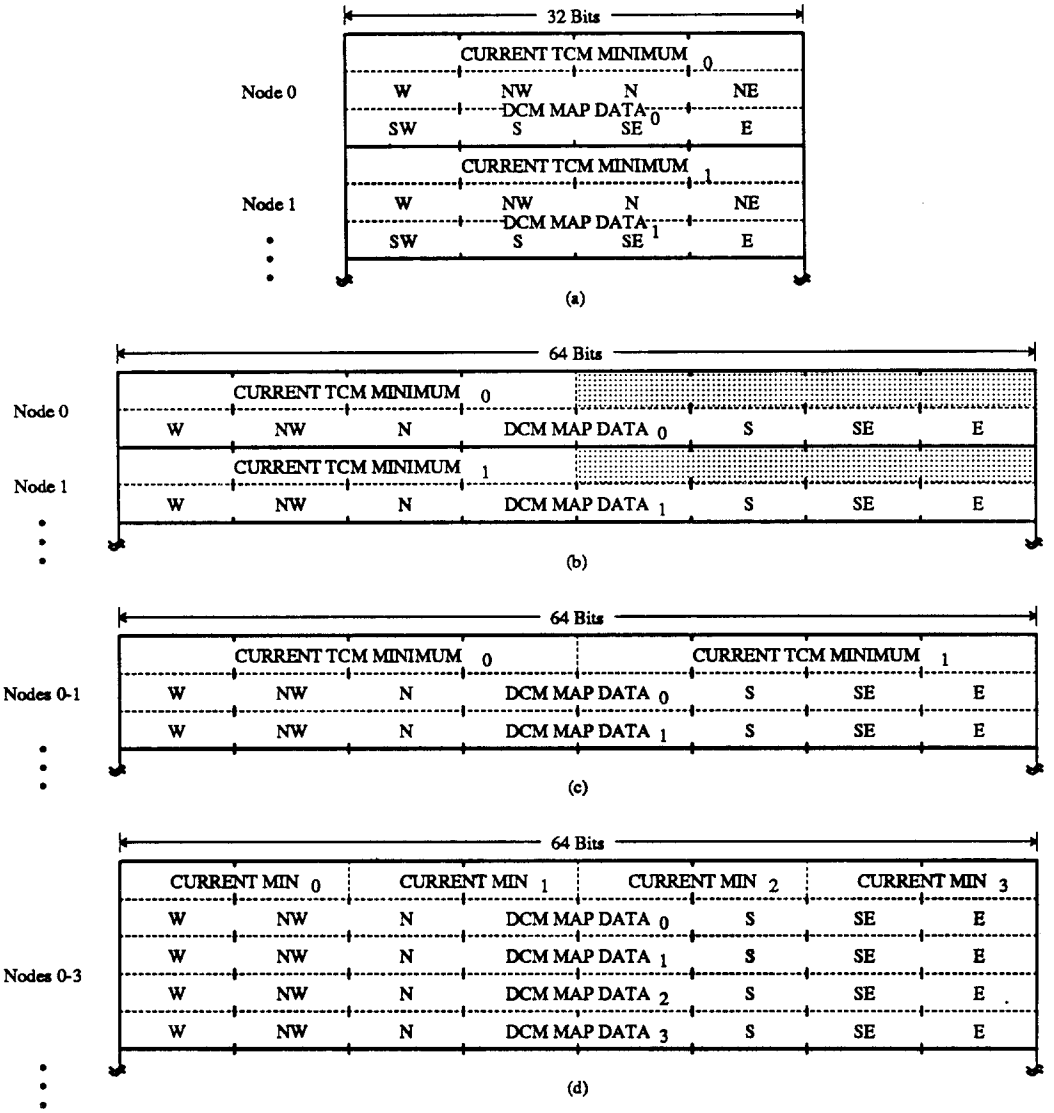


Figure 5-21. Data Structure for Path Optimization Coprocessor.

This figure illustrates four possible data structures for a coprocessor implementation of the path optimization function. The data related to a particular TCM value are stored near it. Neighbor items are stored in sequence in row-major order. Part (a) is a straight-forward layout based on a 32-bit word width using 12 bytes per node. The current TCM minimum value could occupy less than 32 bits, leaving space for a minimum_direction pointer. If addressing simplicity is needed, space can be wasted as shown in Part (b) as long as caching behavior does not suffer as a result. Again, the current TCM minimum could be constrained to 16 bits for the hardware, as explained in Section 5.5.2.4. This consumes 16 bytes per node. Part (c) compacts memory slightly, resulting in a 25% reduction in the amount needed compared to Part (b), or 12 bytes per node as in Part (a). Part (d) reduces that further to 10 bytes per node by compacting TCM entries into one long word, using 16-bit values, and ignoring the minimum_direction pointer for each node.

Table 5-11. Effective Address Calculation for Coprocessor Data Structure									
TCM Cell	Cell Row & Col		Cell Base Address in Memory			Neighbor Offset			
CC	[row]	[col]	(row) * BPR	+	(col) * BPE	K0			
W	[row]	[col-1]	(row) * BPR	+	(col-1) * BPE	K0	-	BPE	
NW	[row-1]	[col-1]	(row-1) * BPR	+	(col-1) * BPE	K0	-	(BPR+BPE)	
N	[row-1]	[col]	(row-1) * BPR	+	(col) * BPE	K0	-	BPR	
NE	[row-1]	[col+1]	(row-1) * BPR	+	(col+1) * BPE	K0	-	(BPR-BPE)	
E	[row]	[col+1]	(row) * BPR	+	(col+1) * BPE	K0	+	BPE	
SE	[row+1]	[col+1]	(row+1) * BPR	+	(col+1) * BPE	K0	+	(BPR+BPE)	
S	[row+1]	[col]	(row+1) * BPR	+	(col) * BPE	K0	+	BPR	
SW	[row+1]	[col-1]	(row+1) * BPR	+	(col-1) * BPE	K0	+	(BPR-BPE)	

Note: *BPR ≡ bytes per row, BPE ≡ bytes per element*

This table shows how the row index and column index of the TCM are used in calculating near-neighbor addresses for data access. The base address of a particular element in the TCM (identified by its [row][col] address) must be calculated, and the offsets to other elements in memory are at pre-computed fixed offsets from that address. CPU registers can be dedicated to these constants, providing a simple R(base)+R(offset) load and store addressing. After determining the address for the first row, other row element base addresses are determined by simply adding the element size as an offset.

For scan-based algorithms, there is a trade off between how many near-neighbor cells are checked each sweep and the number of sweeps needed to converge. The only requirement is that all neighbors be considered before determining that a cell has reached its minimum value. This allows for a simplification to some algorithms. For example, if we modify an algorithm slightly by only loading and checking the W, NW, N, and NE neighbor TCM values, only half the work per cell check is done. Figure 5-23 illustrates two such modified algorithms. Algorithm No. 10 is identical to Algorithm No. 7, except that only four NNR cells are considered for each cell check, reducing the amount of work done on each scan by half. Algorithm No. 11 is identical to Algorithm No. 4, but again the number of cells checked each scan has been reduced to the ones shown. The amount of work per sweep is reduce by half, at the cost of needing to do a few more sweeps to converge. Table 5-12 and Table 5-13 show results of running the algorithms, comparing the number of sweeps for convergence and how well they perform relative to Dijkstra's Algorithm.

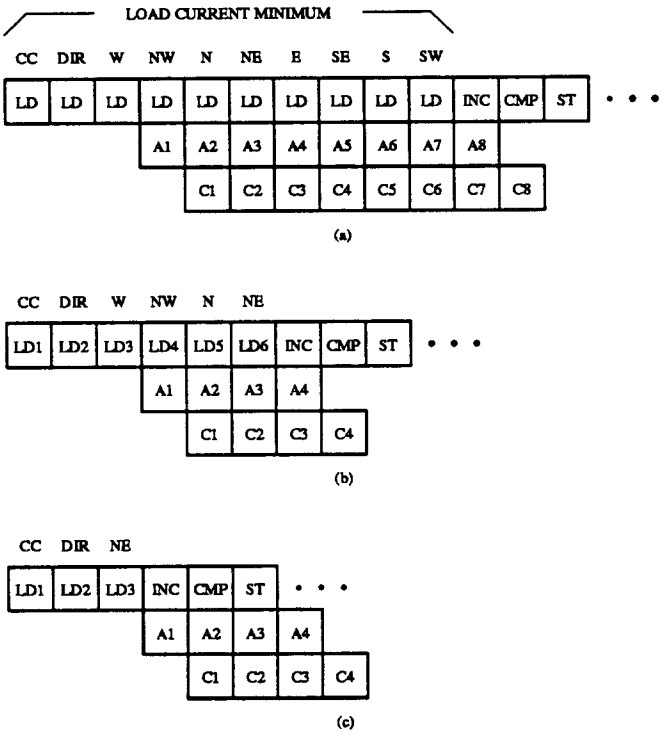


Figure 5-22. Coprocessor Pipeline Stages.

This figure illustrates the instruction flow and corresponding pipeline sequence for the SPUR architecture with a path optimization coprocessor. The CPU performs effective address calculation and operand load/store, while the coprocessor simultaneously decodes the instruction and either performs a load or store as directed by the instruction. The coprocessor performs the add, comparison, and select minimum functions in pipeline fashion similar to the SPUR CPU and FPU architectures.

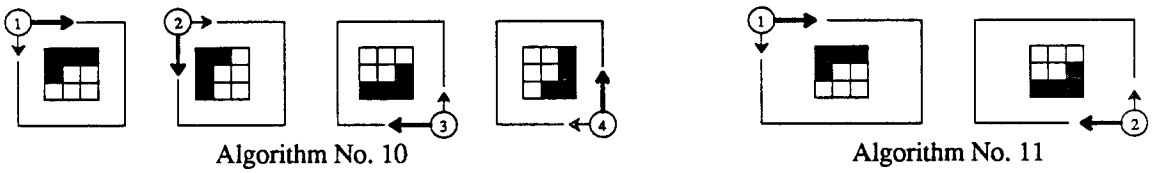


Figure 5-23. Modified Scan-based Algorithm Sweeping Techniques.

The scanning process for each technique is illustrated with a collection of numbered circles and light or dark arrows, as was shown earlier in Figure 5-6. The shaded boxes shown indicate which neighbor cells are considered for the calculation (the unshaded boxes are not involved, reducing the computation time by 50%).

Table 5-12. Scan-based Algorithm Nos. 10 and 11 vs. Dijkstra's Algorithm
(Goal points refer to Figure 5-9, simulated for N=512>)

Data Set and Goal Coords	Ratio of Scan-based Algorithm to Dijkstra's Algorithm			
	Dijkstra	Alg No. 8	Alg No. 10	Alg No. 11
North SF Bay, #3 [150,320]	1.0 (36.6)	3.1	2.2	12.1
South SF Bay, #7 [341,300]	1.0 (36.8)	1.8	2.1	9.8
Sin(x)/x, #6 [085,341]	1.0 (36.2)	1.8	2.2	15.6
Random, #3 [001,003]	1.0 (39.2)	59.8	102.7	94.4

This table summarizes the performance characteristics of Algorithm No. 10 and Algorithm No. 11 for TCM size 512-by-512 and various goal points. We have included Algorithm No. 8 from Table 5-4 for comparison. These represent the ratio of execution time for each algorithm normalized to Dijkstra's Algorithm for the particular data set.

Table 5-13. Performance for Algorithm Nos. 10 and 11 on Random and Terrain Data
(Simulated for N=512)

Data Set and Goal Coords	Number of Sweeps for Convergence		
	Alg No. 8	Alg No. 10	Alg No. 11
North SF Bay, #3 [150,320]	8	8	57
South SF Bay, #7 [341,300]	4	8	46
Sin(x)/x, #6 [085,341]	4	7	71
Random, #3 [001,003]	190	269	256
Ratio of Number of Sweeps to Converge — Random Data vs. Terrain Data			
Random vs. North SF Bay	23.8	33.6	4.5
Random vs. South SF Bay	47.5	33.6	5.6
Random vs. Sin(x)/x	47.5	38.4	3.6

This table shows how scan-based Algorithm Nos. 10 and 11 compare for random and real data sets. Algorithm No. 8 has been included from Table 5-5 for comparison. As with the other scan-based algorithms, the number of sweeps to converge is generally much larger for random data than the terrain data. Note that for random data, Algorithm No. 10 requires 40% more sweeps than Algorithm No. 8, its more complicated counter part. Nevertheless, it is superior, since each sweep requires only half the time. Although Algorithm No. 11 is not as efficient as Algorithm No. 10, it is interesting to note that it takes the same number of sweeps to converge as Algorithm No. 4, which does twice the work per sweep. On that basis, we can say it improves performance by a factor of two for Algorithm No. 4 to make a simpler near-neighbor comparison test.

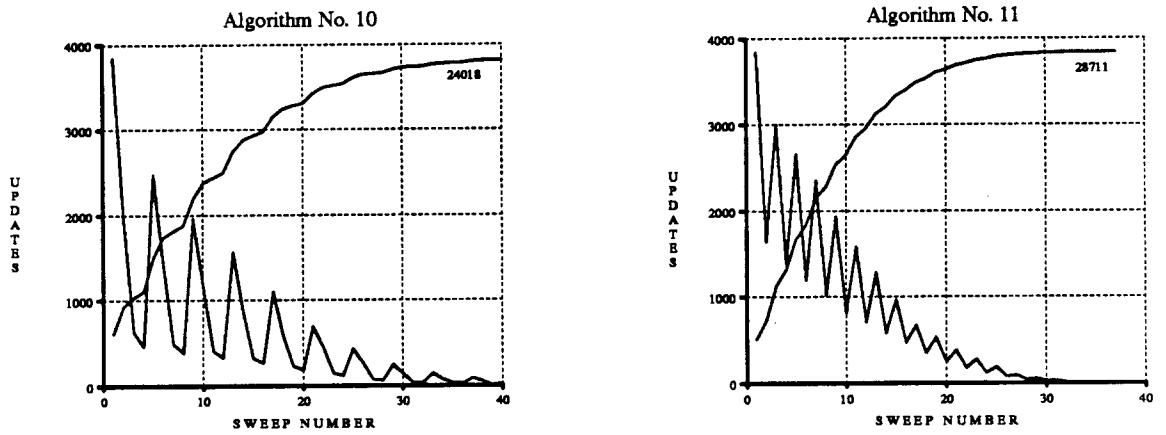


Figure 5-24. Average Updates per Sweep for Algorithm Nos. 10 and 11.

This figure shows the average number of updates per sweep versus the sweep iteration number for Algorithm No. 10 and Algorithm No. 11. The goal point was located at [001,003] for all simulations. As explained in Figure 5-14, the amount of useful work accomplished per sweep is a function of where the goal is located and the scanning/sweeping directions. Here it is important to recognize that each sweep requires half the amount of work of the sweeps illustrated in Figure 5-14 because only half the near-neighbor cells are being checked each time. The overall convergence rate is typically faster when compared to the 8-way versions (Algorithm No. 8 and Algorithm No. 4, respectively).

For Algorithm No. 10 and No. 11, more sweeps will be required to converge than when checking all eight neighbors, since minimized TCM cells determined on a *previous* sweep will not be include in the *current* sweep. This also necessitates two complete sweeps without change before convergence is reached. Although more sweeps must be completed for convergence, the amount of work per sweep is less, and overall, less time will be needed. From Table 5-12 and Table 5-13 it is clear that Algorithm No. 10 is the most effective using the strategy of trading more sweeps for less work per sweep, decreasing the amount of work overall by nearly a factor of two for random data. For terrain data, the number of sweeps is between one and two times what it would be if all near-neighbor cells were compared. If the same number of sweeps is used, the execution time is half what it would otherwise be. If twice as many sweeps are necessary, it will take the same amount of time to converge. Thus, the amount of time needed for the TCM to converge can be reduced as much as half when reducing the number of NNR cells to include in the comparison test.

The architectural simplification resulting from using Algorithm No. 10 (fewer pipeline stages per cell check) is illustrated in Figure 5-22(b). With such a scheme, a cell check nominally consumes nine cycles, a reduction of four cycles over the previous case. At 100 nanoseconds per cycle, this translates into approximately 0.235 sec per sweep, about two-thirds the amount needed for the architecture designed for an 8-neighbor test as illustrated in Figure 5-22(a).

Finally, after a given cell check, most of the data needed for the next cell check is already resident in the coprocessor. The new center cell must be loaded, along with its associated DCM values and its NE neighbor. The most recently minimized center cell value becomes the W neighbor. Hence, only three loads and a shift are necessary to begin a new calculation, and the shift can occur simultaneously with the NE neighbor load. Internally, the path planning coprocessor maintains a 4-deep queue for the TCM current minimum information as it flows through the device. This reduces the software pipeline to that shown in Figure 5-22 (c). By reducing the kernel function to six cycles, the sweep time decreased to 0.156 sec. For terrain data, this results in a convergence time of 0.7 to 1.6 seconds, near the one-second goal. Figure 5-25 shows a simplified hardware block diagram corresponding to the pipeline sequencing shown in Figure 5-22 (c).

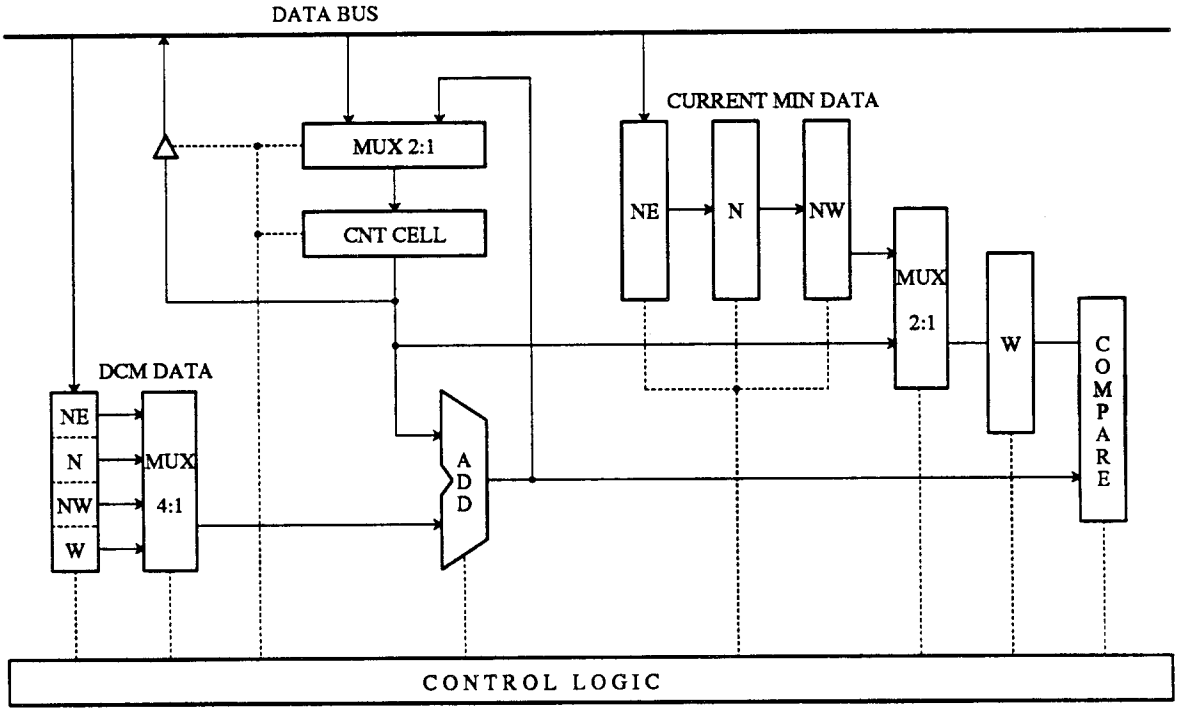


Figure 5-25. Block Diagram of Path Optimization Coprocessor.

This figure is a simplified block diagram of an implementation for a path optimization coprocessor. The timing of load/store and other interactions between the CPU and the coprocessor are based on the SPUR coprocessor interface protocol described earlier in Chapter 4 and detailed in Appendix A.

5.7. Chapter Summary

This chapter has considered the application of dynamic programming algorithms and techniques coupled with coprocessor architectures to the problem of path optimization. Specifically, the shortest path from all points in a grid representing a rectangular region of terrain has been studied.

The range of performance for various algorithms and architectures spanning the spectrum from strictly sequential software to highly parallel hardware implementations have been simulated and their performance determined. The performance characteristics of each of the architectures is illustrated in Figure 5-26.

With cost as one dimension of comparison, a cost-speed plot showing lines of constant performance per unit cost provides a means of comparing cost-effectiveness for different configurations. In Figure 5-27 the various implementations studied are compared. Although there is a broad range of hardware costs, this figure shows that there are roughly three *bands* of equal cost performance. At the low end of the spectrum are purely software implementations, as expected. The VAX family and IBM mainframe offer a level of cost-performance roughly between $10 and $100 per cell check per second. SUN and other microprocessor based systems improve on that performance by roughly one order of magnitude: between $1 and $10 per cell check per second.

Significant performance improvement is obtained only through the use of some form of special architecture or special purpose hardware. In these cases, as illustrated in Figure 5-27, between two and three orders of magnitude improvement over the microprocessor based solutions is achievable.

In certain circumstances, this comparison is of little or no importance. When a performance capability is needed with no particular regard to the price, it is clear that the only way to provide more than 1,000,000 cell checks per second is at great absolute expense. Even so, the cost of special purpose hardware solutions is considerably less expensive when viewed in the perspective of greatly reduced performance provided by the alternatives. This reinforces the premiss that the most cost effective systems are often a combination of specialized hardware elements in conjunction with conventional but

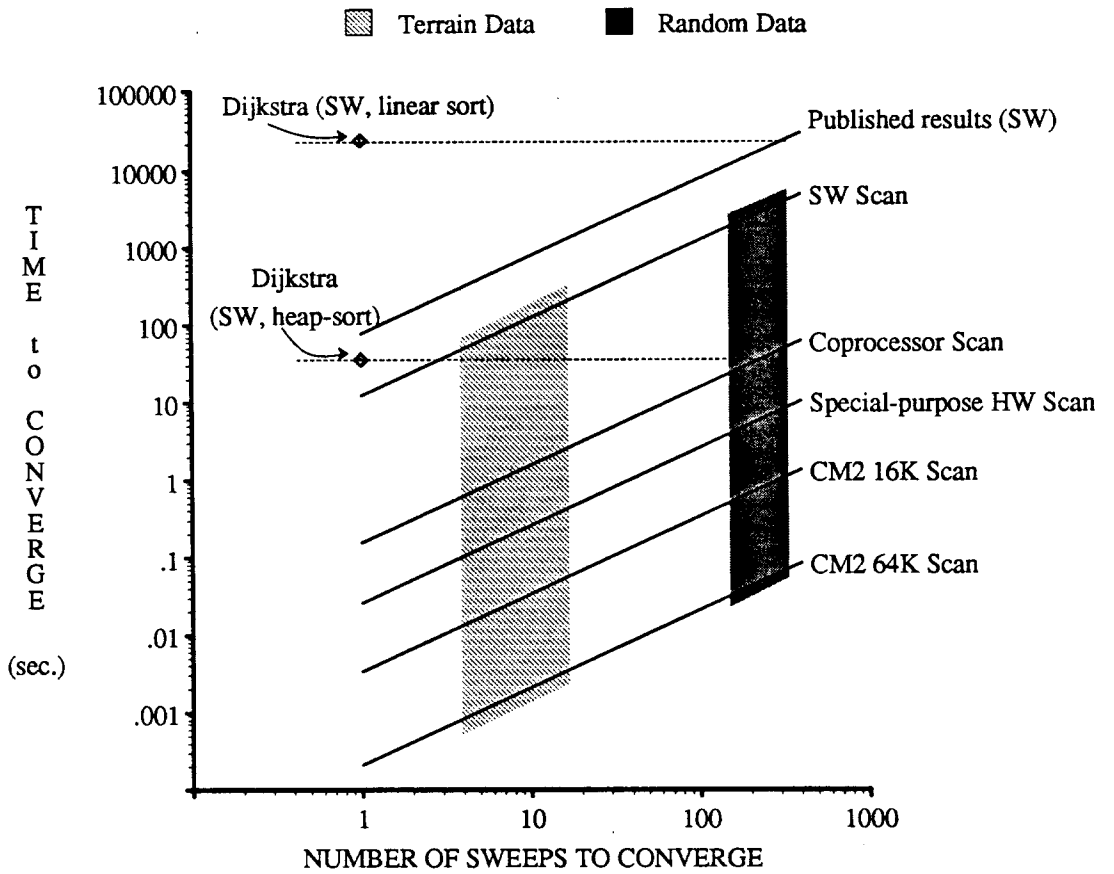


Figure 5-26. Performance of Various Implementations for Path Optimization.

This figure illustrates the difference in performance for several realizations of the path optimization function in software, a kernel-function coprocessor, special dedicated hardware, and a parallel connection-array architecture. Industrial quality software is roughly three orders of magnitude slower than special purpose hardware, and between one and two orders of magnitude slower than the sub-second performance goal. However, the combination of a limited function coprocessor and optimized software reduces that gap to slightly less than one order of magnitude over a broad spectrum of data, and comes very close to meeting the performance goal for the terrain data sets.

streamlined computer architectures that do not get in the way of but rather enhance the capability of the dedicated processing element.

In conclusion, the path optimization coprocessor fits well in the SPUR coprocessor model. The ability to load and store operands during coprocessor operations is a key element to achieving the required performance. The large data cache in a SPUR node provides very high hit ratios for the data structure used in the proposed implementation. The architecture of the coprocessor follows the pipeline scheme of the SPUR CPU and FPU and provides the same concurrency benefits.

This coprocessor implementation uses a simple architecture coupled with an efficient data structure to achieve good run-time performance, orders of magnitude better than pure software implementations. We have shown that the general notion of a hardware coprocessor for path optimization is an attractive alternative and that the SPUR coprocessor model adapts well to requirements of the processing function.

A heapsort coprocessor to implement Dijkstra's Algorithm and/or the application of a path optimization coprocessor to three-dimensional shortest path algorithms used in robotics may prove to be fruitful areas for further research.

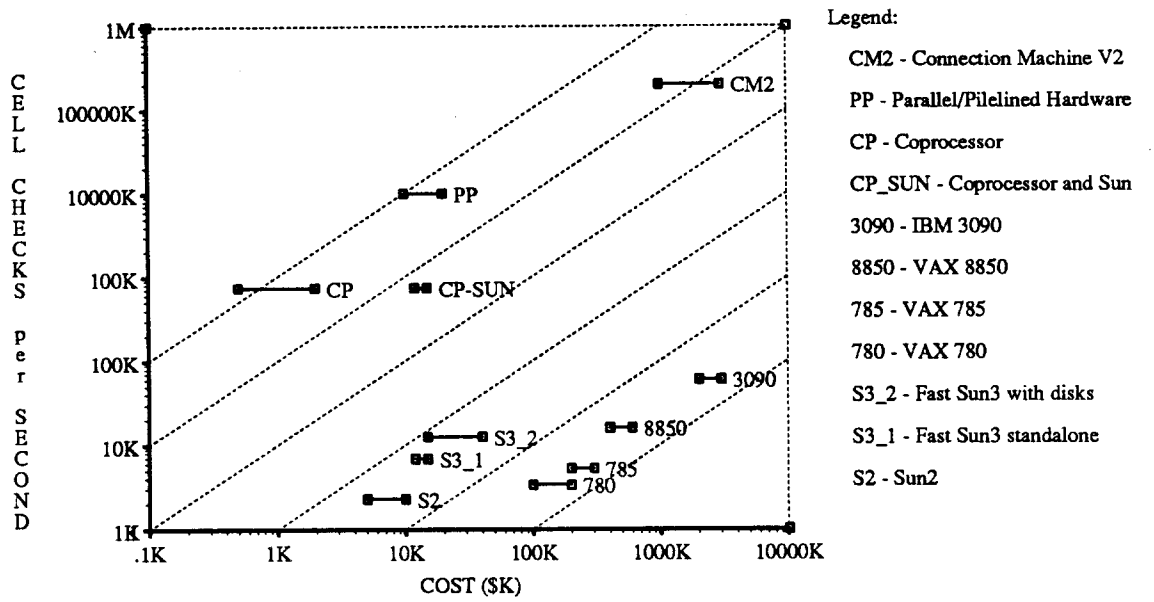


Figure 5-27. Cost-Speed Comparison of Path Optimization Implementations.

This figure illustrates the cost-speed characteristics of the path optimization implementations studied in this chapter. The line for each configuration represents the range over which costs can vary. For example, a Sun3 can cost between $12,000 and $40,000 depending on the amount of memory, the number of disks in the system, the speed of the CPU, and so forth. The sloped dotted lines indicate regions of constant speed per unit cost. For instance, the line passing through the VAX 780 and IBM 3090 is approximately $20 per cell check per second, whereas the special purpose parallel hardware is $.001 per cell check per second.

6 Signal Processing, Vector Floating-point, and Language Coprocessors

6.1. Introduction and Overview

Computer systems are designed to execute well over a wide range of applications. For VLSI microprocessors, the instruction sets are optimized to yield the best cost-performance over that range, under the constraint of finite resources — the amount of silicon that can be put to practical use. To increase the power and efficiency of such systems, execution elements for specialized applications are often added.

In previous chapters, we examined in detail two widely contrasting applications of coprocessors in general purpose computer systems. Arithmetic is a natural part of any CPU instruction set, and a coprocessor designed for floating-point operations would naturally be closely coupled to the operation of the CPU. The operands are relatively large in size and the operations can be relatively long in time. The collection of the data can vary in quantity from incidental scalars to large, vast arrays of numbers.

On the other hand, the functions associated with path optimization are largely data-intensive, and often find realization in non-closely-coupled implementations. Operands are small in size and operations are relatively short in time — but there are many of them. The number of operations over time is large while the quantity of data is relatively large but of fixed size.

In this chapter, we consider other applications that benefit from the inclusion of special purpose hardware in a general purpose computer system to enhance certain functions. We briefly explore and develop the central issues of:

- signal processing coprocessors based on the SPUR FPU model,
- extensions to the current SPUR floating-point coprocessor to accommodate vector operations,
- and a language-specific coprocessor for Prolog that uses a modified SPUR CPU and FPU model.

6.2. A Signal Processing Coprocessor for SPUR

Analog systems use the fundamental properties of physical components, such as resistance, capacitance, or inductance, to realize mathematical relationships between physical world phenomena in the form of signals — current, voltage, or pressure. For example, the voltage build up across a capacitor is the integration over time of the current effectively flowing through the capacitor. Likewise, differentiation, division, addition and summation can be realized. Until faster technology and better architectures made it possible, digital solutions emulating these physical mathematical relationships were simply too slow for anything other than non real-time analyses or simulations. Analog-to-digital and digital-to-analog converters with sufficient resolution, speed, and accuracy were needed to form the bridge between the two worlds. Once the conversion is made, the mathematical relationships are realized explicitly on the digitized signals with techniques and algorithms know collectively as *digital signal processing* (DSP). The current state of technology now provides us with inexpensive devices that can operate at the necessary rates.

The mathematical basis for processing of discrete signals using linear filters began in the early 1600's [Kais67]. Techniques and mathematical investigations dealing with finite differences evolved at the same time that classical numerical analysis and calculus came into being. Early research dealt with idealized, noise-free systems. However, techniques have evolved since 1940 that focus consideration on practical realizations in a non-ideal environment.

Contemporary digital signal processing is coming to rely more and more on floating-point computation [Andr88], circumventing some of the problems associated with overflow, scaling, gain control, dynamic range, precision, and so forth. As shown in Chapter 4, some floating-point programs can be characterized by small kernel operations, which can be optimized for speed and efficiency. Many such processing functions, like equation solving and simulation, have no particular time constraints, though it is usually a matter of making the process *as fast as possible*, or at least *no slower than tolerable*.

The signal processing world divides itself into two camps: non real-time, or simulation oriented, and real-time. As will be shown in Section 6.2.2, the data rates of real-time systems generally preclude the use of general purpose processors to do the main computational loops. In these cases, coprocessors designed and applied to real-time applications are much more like array processors. DSP applications are often unique for that reason, since many of them must be performed in real-time. The question then becomes: Is the SPUR system with its current interface to a floating-point coprocessor sufficient to implement real-time DSP algorithms? If not, what can be done to improve that performance, either by way of architectural innovations or other specialized coprocessors? In the following sections, we examine some of the algorithms and applications of digital signal processing to help determine the answer to those questions.

6.2.1. Signal Processing Algorithms

The Fourier transform is the basis for many signal and image processing algorithms. A comprehensive discussion of the fundamentals and examples of its widespread use are contained in [Brig74, Brig88]. In this section, we present the mathematical basis for signal processing and some of the basic algorithms used.

The Fourier integral is defined by the expression:

$$H(f) = \int_{-\infty}^{+\infty} h(t) e^{-j2\pi ft}\, dt \qquad (6\text{-}1)$$

where $h(t)$ is a continuous function in time and $H(f)$ is a continuous function in frequency. In order to process the signal in the discrete time domain, $h(t)$ must be sampled to yield $h(nT)$, T being the sample period. However, aliasing will occur if the frequency spectrum is not constrained such that $H(f) = 0$ for some $f > f_c$. Practically speaking, this means that the highest frequency component of the sampled signal must be at most half the sampling rate, according to Nyquist's theorem, which states that $T \geq \frac{1}{2f_c}$ to theoretically retain all the information present in $h(t)$.

A second problem comes from truncation of the original signal into a finite number of N samples upon which the transform is performed. Sampling in time creates a periodic frequency representation. Similarly, sampling in frequency creates a periodic time representation. The net result is the discrete Fourier transform (DFT), an approximation to the continuous Fourier transform, which can be expressed as:

$$G(\frac{n}{NT}) = \sum_{k=0}^{N-1} g(kT) e^{\frac{-j2\pi nk}{N}} \tag{6-2}$$

for n = 0,1, ... , N-1.

The most important DFT properties are convolution and correlation. Mathematically, discrete convolution of an input signal x with the system transfer function h can be represented as follows:

$$y(kT) = T\sum_{i=0}^{N-1} x(iT)h\Big[(k-i)T\Big] \tag{6-3}$$

Discrete convolution requires both $x(kT)$ and $h(kT)$ to be sampled and periodic functions with period N. The scaling is necessary due to sampling. Notationally, this is usually written as

$$y(kT) = x(kT) * h(kT) \tag{6-4}$$

Discrete correlation can be expressed as:

$$z(kT) = \sum_{i=0}^{N-1} x(iT)h\Big[(k+i)T\Big] \tag{6-5}$$

sampled and periodic functions with period N.

The fast-Fourier transform (FFT) is a computationally efficient way of evaluating the DFT. Equation (6-6) is the same as Equation (6-2), but with kT replaced by k and $\frac{n}{NT}$ replaced by n:

$$X(n) = \sum_{k=0}^{N-1} x_0(k) e^{\frac{-j2\pi nk}{N}} \tag{6-6}$$

with n = 0,1, ... , N-1. If we let $W = e^{\frac{-j2\pi}{N}}$, Equation (6-6) can be expressed as:

$$X(n) = W^{nk} x_0(k) \tag{6-7}$$

the compact form for a matrix representation. We note that $W^{nk} = W^{nk\ mod(N)}$, and for $N = 4$ for example, the matrix form of the FFT is:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 \\ 1 & W^2 & W^0 & W^2 \\ 1 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix} \tag{6-8}$$

In the form above, N^2 complex multiplies and $N(N-1)$ complex adds are necessary. An easier way to realize the same thing can be accomplished if the W matrix is factored to yield the *FFT butterfly* form, with order $N\log_2(N)$ complex multiplies and adds. This represents the data-flow of the computation. The key to efficiency is factorization of the butterfly form, which results in many variations for special cases. Some of these special forms have been implemented in VLSI at Berkeley and will be described next.

6.2.2. Signal Processing Applications

The ratio of data sampling rate to processing rate provides an indication of how well a computing system is matched to the processing requirements of the computation. For contemporary standard MOS processes, 10 to 30 MHz clock rates are feasible. For normal audio band signals, sample rates between

8 KHz and 44 KHz span the spectrum of applications from telephony to digital audio [Ruet86]. As an example, for circuits clocked at a 10 MHz rate, the ratio $R = \frac{F_{clk}}{F_{sample}}$ ranges between 227 and 1250. Thus, R indicates how much processing can be done between samples.

In the case of special purpose hardware, R indicates how many times each piece of hardware can be used to do different operations in the algorithm. For a general purpose processor, it indicates the number of instructions or clock cycles available between samples. Clearly, as R approaches 1, more parallelism is required to meet real-time processing needs. For video or image processing, the volume and rate of the data is roughly two to three orders of magnitude greater than audio signals, about one processing cycle per signal sample. Clearly, massive pipelining and/or parallelism is needed to do image processing and is far beyond the real-time capability of a general purpose computer system and the scope of this dissertation, and will not be covered further. (Interested readers are referred to [Ruet86] which describes work done at Berkeley with special purpose image processing devices.)

6.2.2.1. A Speech Recognition System

A template-based speech recognizer system developed at Berkeley by [Kava86] is an example of a special purpose signal processor designed to meet real-time constraints. It utilizes custom VLSI components embedded in a conventional UNIX workstation. The recognizer is trained for a particular user's voice characteristics, and is capable of recognizing about 1000 unique words, with a minimum response latency of less than 0.5 seconds.

The speech recognizer consists of multiple special purpose chips for both log and linear multiple channel filtering and pre-emphasis, dynamic programming recursion, gain-normalization, euclidean distance calculations, summation, and so forth. The processing tasks required to do this in real-time can be characterized as follows:

$$\frac{1000\ templates\ x\ 25^2\ \frac{equations}{template}}{300\ milliseconds} = 2{,}083{,}333\ \frac{equations}{second} \qquad (6\text{-}9)$$

For conventional processors, this much computation amounts to roughly 100 million instructions per second [Kava86], or about two orders of magnitude slower than needed for real-time. To meet this processing rate, industry has provided a wealth of specialized programmable DSP chips with architectures designed to implement many of the standard signal processing algorithms effectively. Some of these are listed in Table D-1 in Appendix D. At present, there is only one commercially available DSP chip that provides floating-point arithmetic, the DSP32 manufactured by AT&T. The performance of this chip is roughly 8 MFLOPS [Brod85]. More will be said about its architecture and capabilities in Section 6.2.4. The complexity of speech recognition algorithms exceeds even the most capable single chip DSPs coming to market, and is included here only to place the performance requirements needed for such a complicated processing task in perspective with the current technology.

To determine how well the spur architecture meets DSP applications and compares to the DSP32 implementation, the next sections illustrate the performance of SPUR on three micro-benchmarks.

6.2.3. Signal Processing Benchmarks and Evaluation

To evaluate the effectiveness of the SPUR system applied to signal processing, we selected three typical signal processing benchmark programs from three application areas: spectral analysis, speech generation, and spectrum shaping.* Spectral analysis uses DFT and FFT techniques in performing convolution and correlation. Speech generation involves the functions of linear predictive coding or waveform coding. Spectrum shaping is the filtering of discrete signals with both finite impulse response and infinite impulse response techniques.

*These were recommended by DSP researchers D. Messerschmitt, E. Lee, and E. Wold of UC Berkeley EECS Department.

The generalized mathematical form for all three applications is given in Equation (6-10)

$$y_k = \sum_{n=0}^{N} a_n * x_{k-n} - \sum_{m=1}^{M} b_m * y_{k-m} \quad (6\text{-}10)$$

For some applications, the coefficients are zero, eliminating certain terms and simplifying the overall computation for that application. The basic mathematical operations involved are summation (add/accumulate) and scaling (multiplication), besides the usual house-keeping operations and operands transfers.

In the next three sections, we present each of these applications and report the performance characteristics for various combinations of hardware and software.

6.2.3.1. Spectral Analysis

To implement the FFT, we have chosen a simple version of the Cooley-Tukey [Cool65] algorithm with bit reversal. The set of equations for $N = 4$ is shown in Equations (6-11a), (6-11b), (6-11c)

$$x_1(n_0,k_0) = \sum_{k_1=0}^{1} x_0(k_1,k_0)W^{2n_0k_1} \quad (6\text{-}11a)$$

$$x_2(n_0,n_1) = \sum_{k_0=0}^{1} x_1(n_0,k_0)W^{(2n_1+n_0)k_0} \quad (6\text{-}11b)$$

$$X(n_1,n_0) = x_2(n_0,n_1) \quad (6\text{-}11c)$$

We use a constant filter function with no windowing, and coefficients are computed as needed. The benchmark consists of a 1024-point transform of three sinusoidal signals with frequency ratios 1:3:7, using 32-bit floating-point arithmetic. The results of the FFT were inverse transformed to verify correctness and accuracy. Table 6-1 lists the results of running the benchmark for several processor configurations.

Table 6-1. Performance Comparison of 1024-Point FFT

Host Processor*	Floating-point Implementation	Time (CPU Seconds)	Slow-down relative to real-time
SUN3/160	Software	11.05	475.0
SUN3/160	68881	1.4	60.0
SUN3/160	FPA	0.95	40.0
VAX/8800	FPA	0.6	25.0
SPUR	FPU	0.2	9.0
DSP 32	DAU	0.02	~1.0

This table shows how well several architectures perform a 1024-point floating-point FFT. The CPU time is as reported by the UNIX time(1) command. The real-time requirements come from a sample window of roughly 20 milliseconds, for the assumed sample rate of 44 KHz. The slow-down relative to real-time is the ratio of the time consumed to sample window. Only the DSP32 is able to perform at a real-time rate. SPUR is roughly an order of magnitude too slow.

As can be seen in the table, a conventional workstation or minicomputer using software floating-point routines is wholly inadequate. By improving performance with various floating-point accelerators, there is still a difference of one or two orders of magnitude between the performance needed and what is possible. Only with the specialized DSP32 architecture is real-time processing realized.

*The processor clock rates for this and subsequent tables showing performance comparisons are: 60 nanoseconds for the SUN3, 45 nanoseconds for the VAX 8800, 250 nanoseconds for the DSP32, and 140 nanoseconds for the SPUR system.

6.2.3.2. Speech Synthesis

As the second of three benchmarks, we chose to implement an example of linear predictive coding (LPC). Equation (6-12) represents the computation in the z-transform domain [Honi84].

$$H(z) = \frac{G}{1 - \sum_{i=1}^{M} a_i * z^{-1}} \tag{6-12}$$

We use an auto-regressive all pole model and spectral estimation of the input. We again chose a 1024-point implementation with 50 coefficients and 32-bit floating-point quantities. The computations performed include auto-covariance (largely a dot product inner loop) to produce the matrices and Gaussian elimination to solve for the coefficients. Table 6-2 lists the results of running the benchmark.

Table 6-2. Performance Comparison for 1024-Point LPC Algorithm

Host Processor	Floating-point Implementation	Time (CPU Seconds)	Slow-down relative to real-time
SUN3/160	Software	7.4	320.0
SUN3/160	68881	2.5	107.0
SUN3/160	FPA	1.3	55.0
VAX/8800	FPA	0.6	28.0
SPUR	FPU	0.2	9.0
DSP 32	DAU	0.02	~1.0

This table shows the performance of various architectures on a 1024-point linear predictive coding algorithm, assuming a sample rate of 44 KHz, and 32-bit floating-point arithmetic.

We see that most commercial architectures are still unable to meet a real-time requirements of speech synthesis using the LPC algorithm. The number of coefficients can be reduced by a factor of two to five without a significant effect on quality [Mess87]. The performance of the DSP32 would then meet real-time needs, with SPUR still an order of magnitude less than needed.

6.2.3.3. Spectrum Shaping

The process of filtering signals — extracting some frequencies while rejecting others — is known as spectrum shaping. We chose to implement a low-pass filter, also known as a tapped delay line or finite impulse response filter, given in Equation (6-13) [Oppe75]. We assumed an ideal low-pass filter with with 50 taps and corner frequency at 0.1T.

$$H(z) = \sum_{n=0}^{N} a_n * z^{-1} \tag{6-13}$$

Table 6-3 lists the results of running the benchmark on the selected architectures.

Again we note that other than the dedicated architecture of the DSP32, all conventional processing systems are between one and two orders of magnitude too slow for real-time. The SPUR system comes closest. Using various algorithmic techniques, a multi-processor version of the algorithm could be implemented to achieve real-time performance.

From these three benchmark results, we observe that none of the conventional general purpose architectures tested is able to provide a real-time signal processing capability. As time passes, this will inevitably become possible with faster CPU's equipped with floating-point accelerators. However, the proliferation of specialized chips and in some cases coprocessors will make alternatives more attractive from a cost-performance point of view. In the following sections, we will examine the SPUR architecture in light of these applications to determine how well the current system provides a real-time processing capability, and what might need to change to enhance that potential.

Table 6-3. Performance Comparison for FIR Filter Implementation			
Host Processor	Floating-point Implementation	Time (CPU Seconds)	Slow-down relative to real-time
SUN3/160	Software	3.1	136.0
SUN3/160	68881	1.0	44.0
SUN3/160	FPA	0.30	13.0
VAX/8800	FPA	0.16	7.0
SPUR	FPU	0.08	3.5
DSP 32	DAU	0.013	0.55

This table shows the performance of various architectures on the implementation of a 50-tap FIR digital filter.

6.2.4. A SPUR DSP Coprocessor

When comparing the performance of the SPUR architecture and the DSP32, it is evident that roughly an order of magnitude in improvement in SPUR is needed to achieve real-time operation. From Chapter 4, we found that for tight loops, the SPUR architecture is capable of roughly 0.6 to 1.3 MFLOPS with a 140 nanosecond cycle time (0.8 to 1.8 MFLOPS with the 100-nanosecond cycle time achieved by chips in recent tests). This suggests that floating-point performance of about 10 MFLOPS is needed to process signals in real-time with a SPUR system. What needs to be done in the SPUR architecture to achieve that? To answer this question, we will first briefly examine the architecture of the DSP32. (Please refer to Appendix D for further details.)

The architecture of the DSP32 includes 32-bit data arithmetic (DA) instructions which are highly pipelined and 16-bit control arithmetic (CA) instructions which are not pipelined, except for loads. The DA instructions support various forms of multiply/accumulate between registers, memory, I/O buffers, and so forth. Also, a number of type conversions are included in the instruction set. The CA instructions allow the normal control flow operations (conditional branch, loop counter test, subroutine call and return) and integer arithmetic operations (add, subtract, logic and shift, compare). Because of the decoupled nature of the CA and DA instructions, and the pipelining of the DA operations, test conditions must be established anticipating a typical 3- or 4-cycle latency. This makes programming the DSP32 rather difficult [Andr88]. Table 6-4 summarizes some of the features of the DSP32 architecture and briefly compares it to SPUR.

Table 6-4. Architecture Comparison of DSP32 and SPUR System		
Parameter	DSP32	SPUR CPU/FPU
Control	16-bit special purpose controller, 4 MIPs	32-bit general purpose CPU, 7-10 MIPs
Data	32-bit FPU DAU signal processing format, 8 MFLOPS	32-, 64-, 80-bit IEEE P754 Standard format ~1 MFLOPS
Memory	6 KB on-chip, 56 KB off-chip 250 nsec/cycle	128KB cache @ 100 nsec, main memory, disk
Transfers	Microproc. DMA @ 1.8 MB/sec. 16 MB/sec DAU	10-15 MB/sec. to/from cache
User interface	Assembly, microcode, special purpose development systems	HLL compilers, assemblers symbolic debuggers

This table contrasts some of the architecture features of the DSP32 and SPUR system. The DSP32 is intended for embedded applications and consequently benefits from the development environment typically provided in a general purpose system.

The kernel arithmetic functions of the DSP32 provide the mathematical constructs necessary to implement digital signal processing algorithms efficiently, with little regard for the problems and complexity of programming. The function of the most important instruction implemented in the DSP32 is given in Equation (6-14) using its assembly language syntax:

$$*r_1{+}{+} = a_0 = a_0 + (*r_2{+}{+}) + (*r_3{+}{+}) \tag{6-14}$$

Functionally, this instruction provides the following operations:

- multiply the contents of memory locations pointed to by registers r_1 and r_2,
- add this product to the contents of accumulator register a_0,
- store the result in a_0 and the memory location pointed to by register r_1,
- increment the values in r_1, r_2, and r_3 to point to the next source and destination memory/register addresses.

It is interesting to note that these operations are essentially the same as those implementing the microbenchmarks of Chapter 4, as shown in Table 6-5.

Table 6-5. Instruction Capabilities for DSP32 and Conventional Architecture

Instruction/code	Dest	Src_1	Op_1	Src_2	Op_2	Src_3	Other
DSP32 Instruction	pointed to by r_1, A_0	a_i	+	pointed to by r_2	*	pointed to by r_3	Increment r_1, r_2, r_3
GE	x_i	x_i	+	k	*	y_i	Increment i
DP	p	p	+	x_i	*	y_i	Increment i
PE	p_i	c_i	+	k	*	p_{i-1}	Increment i

The operation of one DSP32 instruction provides all the functionality of several assembly language instructions for conventional computer architectures. This has two positive side effects. First, there is considerable savings in code size. In the case of rolled loops, it makes little difference. However, for unrolled loops, the amount of additional storage required can potentially have a detrimental effect on caching behavior in a conventional architecture. Second, and perhaps most significantly, all the operations specified in Equation (6-14) effectively can happen in one logical time unit because the architecture of the DSP32 is pipelined. The current implementation has a 250 nanosecond cycle time, with two floating-point operations per clock tick, for a maximum performance of 8 MFLOPS.*

The first step to improving SPUR performance is to provide concurrent operation of both the multiply and add execution elements.** A floating-point multiply-add (FMAD) instruction is the most often required arithmetic operation. The current implementation supports a 3-cycle floating-point add and 8-cycle floating-point multiply [Bose88b]. The SPUR FPU design can easily accommodate one less cycle per instruction by eliminating register-read and register-write operations for the FMAD instruction, by providing a forwarding path as is done on the SPUR CPU. With an increase in total chip size of about 10%, the multiply/divide unit of the current SPUR FPU architecture could double its performance by retiring more bits per iteration. These two changes combine to yield a FMAD instruction of four cycles. If a full array multiplier is used, a 25% increase in chip area in current technology could reduce the multiply latency to two cycles.

Finally, if the additional control is added to allow pipelining of the function units and chaining is incorporated to allow simultaneous operation of the multiply and add units, the FMAD instruction can effectively be executed in a single cycle. Thus, the performance of a modified SPUR DSP FPU implementing the microbenchmarks discussed earlier is summarized in Table 6-6.

*Rapid advances are being made in the area of monolithic DSP chips that support floating-point arithmetic. The DSP32C, to become available mid-1989, is intended to provide the same functionality but with a cycle time of 80 nanoseconds, suggesting a peak performance rate of 25 MFLOPS.

**A recently announced commercial FPU provides this capability [Rowe88].

Table 6-6. Performance Comparison for SPUR DSP Coprocessor				
Host Processor	Floating-point Implementation	Slow-down relative to real-time		
		FFT (1024 pt)	LPC (1024 pt, 50 pole)	FIR (50 tap)
SUN3/160	Software	475	320	136
SUN3/160	68881	60	107	44
SUN3/160	FPA	40	55	13
VAX/8800	FPA	25	28	7
SPUR	FPU	9	20	3.5
SPUR	DSP	1-2	2-3	0.7-1.2
DSP 32	DAU	1	2	0.55

This table again shows the performance for various architectures and combinations of software and hardware to implement floating-point computation on three signal processing algorithms shown in Table 6-1, Table 6-2, and Table 6-3. The point of interest here is the SPUR DSP FPU. By allowing pipelining and concurrent multiple function units, at a cost of 30% to 40% increase in chip area, an order of magnitude performance improvement is achieved, allowing a general purpose workstation microprocessor based system to perform real-time signal processing applications.

6.2.5. Signal Processing Coprocessor Summary

This section has considered the topic of digital signal processing. We have examined the requirements of various algorithms, both in terms of functionality and real-time performance. We have considered and evaluated several alternatives in implementing these algorithms, including general purpose computers equipped with a range of floating-point capability from software through highly specialized coprocessors. We have contrasted the general purpose system solutions with specialized hardware in the form of a monolithic DPS chip.

We conclude that general purpose computers are largely inadequate for real-time signal processing functions, even if equipped with floating-point accelerator hardware. We have proposed an augmented version of the SPUR FPU for DSP which is generally adequate for such tasks. The advantages of the general purpose SPUR system in terms of the code development cycle, debugging, operating system support functions, and general environment suggest this would be a desirable capability for research in the real-time signal processing field. This is based on the assumption that the current SPUR FPU can be pipelined and made more functionally parallel with current technology.

To summarize, no changes are required in the current implementation of the SPUR coprocessor interface to accommodate a DSP-style coprocessor. The CPU would be kept largely busy with transferring operands, if the SPUR DSP FPU is pipelined. Since the data resolution of signal processing typically requires no more than 32-bit floating-point operands, different front-end pack/unpack logic could be used to receive two operands per load with the 64-bit bus. The model of the SPUR DSP FPU relies on multi-port register files to allow simultaneous transfers and operations. As technology scales, the cycle times will be reduced, and the processing speed for real-time applications will be more attainable. We explore the ramifications of achieving a pipelined capability in the next section, which considers vector floating-point processing in general.

6.3. A Vector Floating-point Coprocessor for SPUR

As discussed in Chapter 1, rapid technological developments are enabling us to roughly double integer CPU performance every year since 1984. This trend is likely to continue for the next five to seven years, at least. To match that performance, floating-point computation must take advantage of lower latency function units and other advanced architecture techniques to keep pace. One technique uses special hardware to process groups of operands as a unit, called a *vector* using what has come to be called *vector processing.*

The basic notion of vector processing is simple: operate on two vectors, element by element, to produce a third vector. The function units designed for such operations are usually pipelined, and the memory system must supply one element from each input vector and store one element of the output result each time unit. Thus the maximum rate is determined by the throughput of the execution element or bandwidth of the memory system. That rate is achieved only when neither the memory system nor execution element causes the other to stall. First, we will consider problems generally associated with the design of the memory system and then the specific application to the SPUR architecture.

6.3.1. Memory System Design for Vector Processors

Historically, the biggest problem for vector processors is the memory system design. Techniques to sustain the flow of data or schedule operations to reduce the flow are typically used to make it work [Ston87]. If the function unit is capable of producing one result every time unit, the memory system must be capable of two operand reads and one result write each time unit as well*. Consequently, memory systems have provided either (1) multiple, independent memory modules allowing simultaneous access or (2) a memory hierarchy, with a small, high-speed intermediate memory for supplying operands and accepting results coupled with a main memory with high-bandwidth transfers between each.

6.3.1.1. Multiple Memory Modules

Multiple memory modules ideally allow partitioning of data so that no access conflicts occur during the sequence of reads and writes — each access in time is to a different module in space and can occur simultaneously. If d time units elapse between a memory module request and its reply, then d separate memory modules must be used to maintain one flow of operands for a pipelined vector operation. If one iteration of the vector instruction requires two operands and supplies one, it will then need $3d$ independent memory modules to maintain conflict-free operation in time.

Even with multiple memory modules, conflicts occur from accessing patterns and stride, not to mention linked conflict that occurs when loading operands and storing results [Buch87]. There are many techniques used to solve this problem, including data skewing, direct storage of diagonals, and so forth [Ande87]. The bandwidth of such memory systems is nevertheless determined by the access time of memory modules, since the execution elements are directly connected, which adds to the delay of the computation. [Ande87].

6.3.1.2. Fast Intermediate Memories

The second method used to support vector operations is fast intermediate memories, which can overcome some of the disadvantages of multiple memories mentioned in the previous section. The Cray-1 XMP, for example, uses as many as four levels of memory of varying speeds between the pipelined arithmetic units and main memory. These include the vector registers, scalar S-registers, the T-registers that serve as a cache for the S-registers, the address A-registers, and B-registers that serve as a cache to the A-registers [Cray86]. These are all managed by software and visible to the user. The transfers are faster than caches, however, since each reference is a programmed register access and no tag-match comparison is needed. A 256-entry instruction buffer is also used. By providing high-speed buffers with the appropriate bandwidth and main memory with multiple ports, overlapping of vector operations is also possible. The output of one vector stream can be routed to the input of another vector operation using *chaining*, allowing multiple function units to operate simultaneously. Other advantages of fast intermediate memories include multiple use of operands, more efficient access patterns than main memory, and data delay buffering [Kogg81, Ston87]

*This ignores the demands of I/O and the instruction stream on the memory system. Since a single vector instruction can specify many operations, instruction fetches are usually negligible when compared to vector operand references.

6.3.2. Memory System Performance Characteristics for Vector Processors

Figure 6-1 illustrates the difference between the Cyber-205, a multiple memory system, and the Cray-1 XMP using intermediate memories and vector registers. The execution time of a Cyber-class machine is predicted by Equation (6-15) [Hock81].

$$r = r_\infty \frac{N}{N_{1/2} + N} \tag{6-15}$$

Here, N is the vector length, r_∞ is the theoretical execution rate for infinitely long vectors, $N_{1/2}$ is the vector length at which the execution rate is half r_∞. Although this predicts preformance of machines that rely on multiple memories and memory-to-memory vector operations, it is inaccurate for vector-register based machines. Bucher and Simmons [Buch87] propose a different model for predicting performance for vector computers, shown in Equation (6-16).

$$t = t_L + \left\lceil \frac{N}{RL} \right\rceil t_s + N t_E \tag{6-16}$$

The execution time is t, N the vector length, RL the vector-register length, t_L the overhead time consumed in starting the pipelining process, t_s the overhead associated with processing each strip of the vector, and t_E the pipe-stage time (usually the cycle time of the execution element or some small multiple). Equation (6-16) is accurate for vectors of all but small lengths. For small vectors, there is a minimum amount of time consumed independent of vector length due to the dominance of scalar house keeping instructions. As shown in Figure 6-1, if the Cray-1 did not incur the overhead per-strip cost t_s, it's average performance would follow the lower of the two dotted-line boundaries of its performance. In the next section, we consider some of the characteristics of programs and their potential influence on vector architectures.

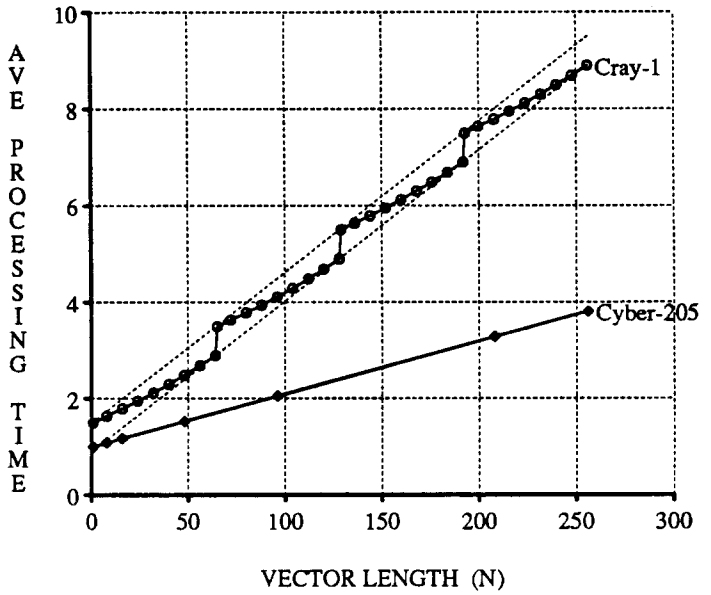


Figure 6-1. Vector Program Execution Time versus Vector Length.

This figure shows the average execution time in microseconds for a simple vector operation — vector addition of a vector plus scalar — for the Cyber-205 and Cray-1 super computers. The Cyber is a memory-to-memory vector architecture, using multiple memory banks to provide the operand access bandwidth needed. The Cray-1 uses high-speed vector-to-vector register operations. Note that for Cray, performance has *discontinuities* at multiples of the vector register length of 64. This is due to the overhead associated with starting another vector *strip*. A strip is simply the size of the vector registers. (Adapted from [Buch87].)

6.3.3. Vector Program and Workload Characteristics

Vector processing benefits from the characteristics of certain programs. Rather than rely on the statistical workings of a cache to speed up the computation by providing often-used data near the function unit, the determinism of the program and data structures used by the program allow the programmer to know what will be needed, when it will be needed, and capitalize on that information. In traditional vector processor designs, caches are not often used and the processor need not concern itself with coherence between cache and memory, which can cause access interlocks or slow the processor down.

Vector and matrix operations are succinct descriptions of voluminous but highly-regular calculations. The requirements on the hardware architecture suggest that easy access to rows and columns of a matrix is important. Besides vector operands, provision for fast access to scalar operands and production of scalar results (for example, minimum value, maximum value, sum) is necessary. Also, since some vectors will necessarily be short in length (as in some phases of LU decomposition or Gaussian elimination), the effects of overhead associated with filling the pipeline must be kept as small as possible. The use of multiple memory modules or high-speed buffering can help in vector and matrix operations.

Sometimes specialization for a particular class of problems is a cost effective solution. For example, with dot product ($P = P + (X[i] * Y[i])$, a function unit with two inputs to a multiplier and one to the adder, with the output of the multiplier being the second input to the adder could be constructed. A problem still exists. Since P is used as an operand following its computation, the pipelining of operations may be long enough for completion of the addition step. This can be overcome by observing that summation is associative. By using the values of the adder as they become available, an additional d time units is expended to generate the final summation of d separate sums.

Now that we have identified some of the general notions associated with vector processing and processors regarding memory and programs, we next explore how compatible the SPUR system is with the requirements of vector processing.

6.3.4. A SPUR Vector Processing Architecture

Fundamentally, vectorization allows the overhead factors we examined in Chapter 4 to be eliminated or amortized over a long sequence of floating-point operations. For example, the process of determining the address of the next operand is often implicit for vectors, and requires no explicit computation in the native architecture assembly language. Also, mechanisms are used that automatically step through arrays with constant stride. The SPUR architecture and floating-point coprocessor do not provide such mechanisms. The question remains: how well will it perform on vector processing?

In the following sections, we briefly consider how the current design of the SPUR memory system, instruction set, and floating-point coprocessor adapt to the requirements of vector processing. We outline changes or additions to the architecture that would be necessary to make it more effective.

6.3.4.1. Memory System

First, with respect to the interaction with memory, the SPUR CPU is in charge of all operand loads and stores which can proceed independently of the floating-point execution unit. For double precision floating-point operands, and the 64-bit wide bus between the FPU and memory, each operand load or store consumes one cycle. For dot product, the operands can be supplied almost continuously if strip mining of the matrix is used and many CPU registers are used as offsets to a base register that is incremented each loop iteration. This affords some decoupling of the memory system from the execution element operation. But, the latency of the execution unit must accommodate this high-speed operand transfer rate in order to be effective. A SPUR multiprocessor system provides a means of emulating both independent, multiple memory modules and high-speed intermediate memories. Each SPUR processor maintains its own 128 KB mixed instruction-data cache. The multiprocessor system capable of assigning processing tasks to several processors and using techniques such as strip mining can prevent access conflict and emulate multiple-memory systems. At the same time, the caches serve as high-speed intermediate memories between the floating-point coprocessor registers and main

memory, offering the advantages of high speed buffers.

The disadvantages of the SPUR memory system architecture come from the potential conflict, or at best inefficiencies, at main memory when emulating multiple memory modules. Although access to operands held in the cache is fast, there is still the cost of the load or store to/from the register file as well as the intrinsic amount of processor cycle time for the cache access, including tag comparison, chip crossings, possible multiplexing, and so forth.

The bandwidth between main memory and the SPUR cache is 14.5 MB per second for the nominal 22 bus cycles per 8-word transfer of 32-bit words with a 100-nanosecond clock period. A dedicated vector-data cache with a wider word width to memory might be necessary to support the processing rate of a pipelined vector unit. Also, conflict between the integer parts of programs and vector operands in the mixed instruction-data cache of SPUR could be avoided by separate caches for each. Within the architecture of a vector-cache memory, multiple banks or interleaving may be used to achieved the needed bandwidth. A balance between the cache-service time and operand use rate must be attained.

6.3.4.2. Instructions

SPUR uses an on-chip instruction buffer for rapid access to frequently used instructions. The buffer holds 128 instructions and is large enough for most small loops. The SPUR instruction set uses 83 of the available 128 opcodes, and includes the floating-point operations mentioned in Appendix A. Consideration for typical vector operations as listed in Table 6-7 to support vector processing would be needed in the instruction set. A more detailed analysis of program characteristics and algorithm issues is needed to determine an adequate set, but is beyond the scope of this dissertation.

Table 6-7. Vector Floating-point Instructions*

Instruction	Meaning
Vector-vector multiply	$Y[i] = X[i] * U[i]$
Vector-scalar multiply	$Y[i] = X[i] * K$
Vector-vector addition	$Y[i] = X[i] + U[i]$
Vector-scalar addition	$Y[i] = X[i] + K$
Vector inner product	$Y[i] = \sum_{i-1}^{N} X[i] * U[i]$
Vector summation	$Y[i] = \sum_{i-1}^{N} X[i]$
Vector sum of squares	$Y[i] = \sum_{i-1}^{N} X[i] * X[i]$
Vector move	$Y[i] = X[i]$
Vector conversion	fixed-point to floating-point, and vice-versa

The implementation of the instruction set could benefit from a decoupling of the control unit of the vector processor from that of the SPUR CPU. Unlike the floating-point coprocessor, the sequence of operations for a vector coprocessor is much more predictable. Consequently, a vector coprocessor can profitably manage its own resources, including data array address determination, sequencing, stride and offset computation, vector-data cache interaction, and so forth. Obviously, provision for handling multiple exceptions during the course of a vector floating-point computation must be provided. Further research is need to determine the amount of hardware needed to do that. The interested reader is referred to [Smit85, Sohi87] who discuss various methods using reservation stations, tag units, and reorder buffers to preserve the effect of an in-order execution model for out-of-order instruction completion and pipelined function units and yet allow accurate exception handling.

*Adapted from [Kogg81].

6.3.4.3. Programs

In the normal SPUR environment, the technique of loop unrolling can be used to achieve some of the advantages of vectorization. For pipelined function units, the depth of loop unrolling is typically equal to the latency of the pipeline. Operands are furnished fast enough to consume the available memory bandwidth until results are produced. Then the memory bandwidth is used for storing the results. In some instances of loop unrolling, it may be advantageous to intersperse arithmetic instructions with load/store or other integer instructions. This is software vectorization to take advantage of a particular floating-point hardware implementation.

In a vector unit capable of managing its own resources and responding to vector instructions, the software optimizations may be at a higher level, such as providing the data storage in a manner compatible with the function of a vector floating-point unit and issuing vector instructions to operate on the data according to the semantics of the computation. Loop unrolling may come into play to vectorize recurrence equations or outer loops of nested-loop computations. Further study is needed to identify the proper choices.

6.3.4.4. Execution Elements

The current microarchitecture of the SPUR FPU provides for a three-cycle floating-point addition and eight-cycle floating-point multiplication. Table 6-8 lists some of the parameters that would possibly change in the current SPUR FPU to accommodate vector execution elements. Section 6.3.1 discussed the effects of vector length versus number of vector registers for the Cray-1 architecture. For vectors of length slightly larger than the number of vector registers, a 20% reduction in performance resulted due to vector-fill time. For this reason, even though it is possible to pipeline the operation to produce one result per clock tick once the pipe fills, a short pipeline is needed to minimize the inefficiencies inherent with short vectors. The point at which the extra area necessary to support faster operations is no significant advantage needs further research beyond the scope of this dissertation. Multiple function units on the same chip would allow chaining operations and multiple independent operations in parallel, which would require multiport register files.

Table 6-8. Parameters for Pipelined and Vector FP Execution Elements

Parameter	SPUR FPU	Vector Unit
Technology	1.6 - 2.0 μ CMOS	0.8 - 1.2 μ CMOS
Transistors	111K	200 - 500K
Chip pins	208	~ 300
Chip area	453 mils x 453 mils 11.5 mm x 11.5 mm	~500 - 600 mils / side ~12.5 - 15 mm / side
Registers	16	64 - 128
Add cycles	3	1 - 2
Multiply cycles	8	1 - 2
Control	10% - 15%	10 - 20%
Power	~1 Watt @ 10 MHz	~3 - 5 Watt @ 20 - 40 MHz
Memory bus width	64-bit	64-128-bit

This table lists some of the design changes that would be necessary to transform the current SPUR FPU to a vector FPU using the same VLSI processing. Further study beyond the scope of this dissertation is needed to identify the optimum choices based on program, algorithm, and implementation details and tradeoffs.

6.3.4.5. Control

The SPUR CPU is implemented as a four-stage pipeline. The SPUR FPU is pipelined only to the extent of being able to initiate back-to-back operations to the execution element; i.e., the second floating-point instruction's decode phase is overlapped with on-going operations. Since the CPU incorporates an on-chip instruction buffer, two possibilities exist for sending instructions to a SPUR Vector FPU. Either parallel instruction decoding is done similar to the current FPU design, or explicit vector-unit commands are sent from the CPU, much like load/store instructions with memory.

The second method may result in up to two more cycles latency in initiating a vector operation, but may be an advantage since it is a more generalized interface and does not require the explicit specification of the operation in the CPU instruction set, just the class. Provision must be made in the instruction set to differentiate vector floating-point operation and scalar floating-point operations, and specify the operation, source operands, destination, addressing or stride information.

Other issues to consider include the generality of a vector unit. If the vector unit is designed to allow vector-integer operations (sometimes used in graphics), then the vector control unit may in fact interact with a pipelined floating-point unit of its own much like the CPU does with the scalar FPU. In this case, a decision as to where the vector registers are and what the format of the data held in them must be considered. For example, an integer-floating-point vector unit may contain a set of N integer-wide registers that can be addressed as N/2 double-precision floating-point registers. Again, nominal vector length, pipeline latency, interface coupling and overhead, the number of execution elements, and so forth all enter into the decision in determining a suitable implementation.

6.3.4.6. Performance

The SPUR FPU is designed to operate as a non-pipelined scalar function unit. Only one operation can be completed at a time, and operations can not be overlapped. In Chapter 4, we determined the processing rate for GE, DP, and PE to range from less that 1.1 to slightly more than 1.8 MFLOPS (millions of floating-point operations per second). Through simulation, we are able to predict the performance of modified versions of the SPUR system for various configurations of software pipelining, hardware pipelining, multiple execution units, separate instruction, data, and vector caches, and so forth. We consider the following configurations:

- Scalar SPUR CPU and FPU system (nominal design)
- Scalar system with faster FPU operations, non-pipelined function units
- Scalar system with pipelined FPU operations
- Vector system with vector-pipelined FPU

with the programs used in Chapter 4: Gaussian elimination, dot product, and polynomial evaluation.

6.3.4.7. Faster Execution Units

By maintaining the interface and architecture as defined for the base-line SPUR design, we found in Chapter 4 that the floating-point performance as a function of execution unit time is as illustrated in Figure 6-2.

Even with substantially faster operation, the non-pipelined scalar FPU makes achieving high performance difficult. In the next section, we consider ways to remove that obstacle.

6.3.4.8. Software and Hardware Pipelining

In Chapter 4, we saw that despite the non-pipelined, scalar design of the FPU, by using the technique of software pipelining (i.e., unrolling loops), some benefits of pipelined execution could be attained. Unrolling the dot product loop twice resulted in earlier cache misses, allowing otherwise wasted time spent waiting for operands to be overlapped with FPU operations. If we go a step further and consider FPU function units with the same latency as the original design and simply allow hardware pipelining and independent units, loop unrolling can have a substantial beneficial effect. The sequence of operations for DP unrolled a depth of four is shown in Table 6-9.

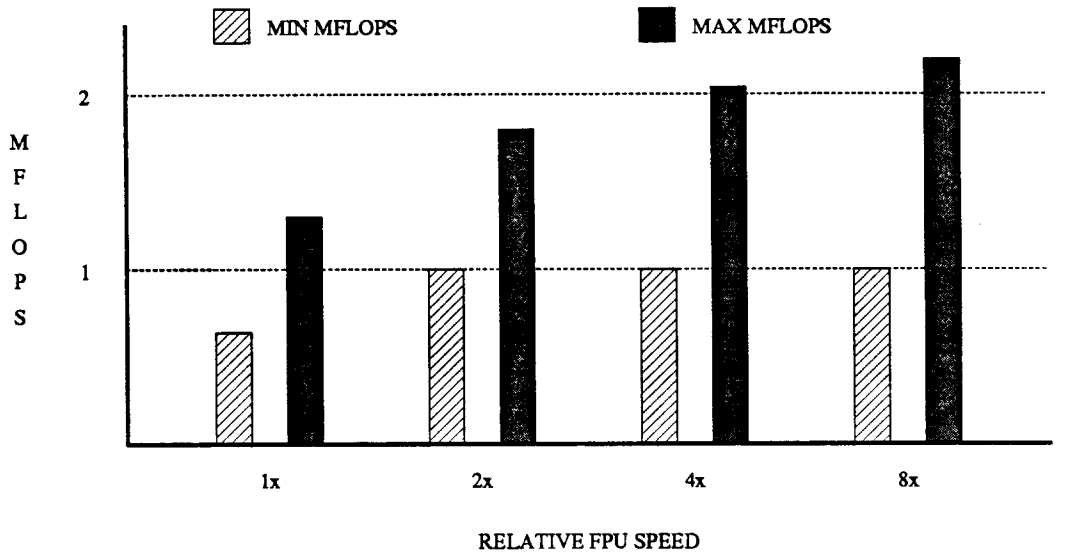


Figure 6-2. Nominal SPUR System Performance.

This figure shows the execution time in MFLOPS for the SPUR system using various FPU execution unit speeds. The speeds are reported as a multiple of the nominal values; i.e., 4X means the intrinsic operations of the FPU are four times faster than the nominal case. There is more than a factor of two variation for a given speed. This stems from the fact that some programs (GE, DP, or PE) had relatively more or less overhead operations versus the number of floating-point operations per loop cycle. As can be seen, a significant speed up in execution unit rate makes little difference in overall performance since it is still non-pipelined scalar execution. It should be noted that these numbers do not include the influence of cache misses.

Table 6-9. Software Pipeline (Unrolled Loop) for Dot Product on SPUR

Instruction Number	Instruction or Operation	Operands or Comment
1	FLD	X(n)
2	FLD	Y(n)
3	FMUL	X(n-1) * Y(n-1)
4	FADD	P + X(n-3) * Y(n-3)
5	FLD	X(n+1)
6	FLD	Y(n+1)
7	FMUL	X(n) * Y(n)
8	FADD	P + X(n-2) * Y(n-2)
9	FLD	X(n+2)
10	FLD	Y(n+2)
11	FMUL	X(n+1) * Y(n+1)
12	FADD	P + X(n-1) * Y(n-1)
13	FLD	X(n+3)
14	FLD	Y(n+3)
15	FMUL	X(n+2) * Y(n+2)
16	FADD	P + X(n) * Y(n)
17	ADD COUNT	Increment loop counter
18	CMP_BR COUNT	Loop test, delay branch
19	ADD OFFSET	Form X, Y base address

The unrolled loop consists of eight floating-point operations and various operand load, store, and integer house-keeping instructions. For a cycle time of 140 nanoseconds, this represents slightly more than 3.0 MFLOPS and is between 2.3 and 5.0 times the scalar FPU rate. (This would be 4.2 MFLOPS at a 100 nanosecond cycle time). The cost in terms of the implementation is 2% to 5% more area for control and additional latches and about 5% to 8% more area for the function units.

One of the problems with unrolling loops is that a certain amount of time must be spent in filling and draining the pipeline in software. This also occurs with vector processing in hardware. Figure 6-3 illustrates the three situations that exist over time with pipelined operations: filling the pipe, the steady-state asymptotic rate, and draining the pipe. With unrolled loops, there are many occasions when odd groups of vectors are created by the vector length modulo the unrolling depth which must be handled by a special piece of code, with some resulting inefficiencies.

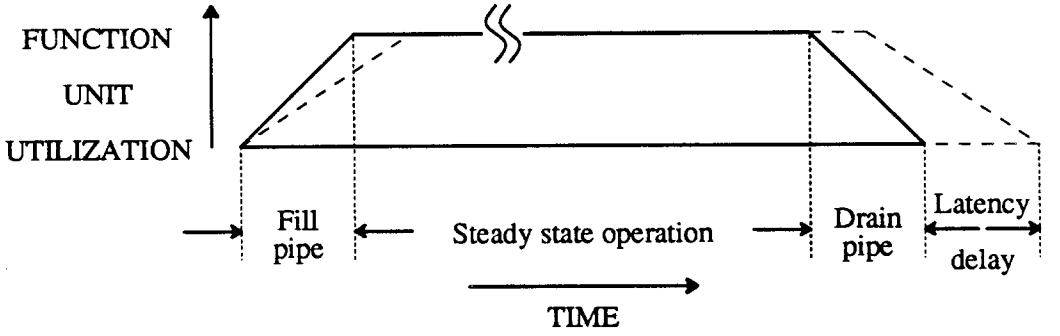


Figure 6-3. Execution Unit Utilization vs Time for Pipeline Operation.

This figure shows utilization of pipelined FPU execution units over time. The start-up costs of filling the pipeline, whether software or hardware, may be insignificant if the steady-state period is long. The latency of the pipelined units determines the slope of the curve, representing filling or draining the pipeline. The shorter the pipeline, the fewer cycles needed to fill the function units to capacity or drain them at the conclusion of a computation. The dashed line shows the situation for filling pipelines that have longer latency. The overall increase in execution time is designated by the latency delay segment.

Once the pipeline is full, the processing rate is identical for both long and short latency function units. The inefficiencies of the pipeline begin to dominate when short vectors or scalars are computed. In the next section, we consider a complete vector unit for the SPUR system.

6.3.4.9. Vector Control and Execution Elements

By combining the features of pipelining in hardware with specialized control mechanisms, full vector processing for the SPUR system can be achieved. Such a system adds the following to the current architecture:

- a vector control unit (separate from the general purpose CPU),
- multiple sets of vector registers,
- multiple pipelined function units,
- chaining control, and
- a vector-data cache.

With the dedicated path between the vector-data cache and vector registers or execution elements, a wider data path allowing two operands per load (128 bits for double precision) is feasible. With a separate vector control unit, two instruction streams operate in parallel, with the vector control unit managing the vector hardware and pipelined floating-point unit. The peak rate, assuming one operation per clock tick for the nominal clock rate of 100 nanoseconds, is 10 MFLOPS. With chaining or an accumulating multiplication unit, the peak rate is 20 MFLOPS.

The constraints of VLSI, in terms of the number of registers, the vector register length, whether the floating-point execution elements reside on the same chip as the registers, the function unit latency,

the interface between the function units, vector-data cache, and the overall control scheme (which includes maintaining multiple condition codes, state for multiple exception conditions, and so forth) are all extremely important and interesting issues that must be evaluated precisely. This all deserves extensive further study, but is beyond the scope of this dissertation. The single-point design briefly covered here is one combination of some factors and represents a significant departure from the current implementation of the SPUR system. This is considered here as an extension and point of contrast to the nominal design discussed in this dissertation. Figure 6-4 illustrates the performance of the various floating-point configurations for SPUR.

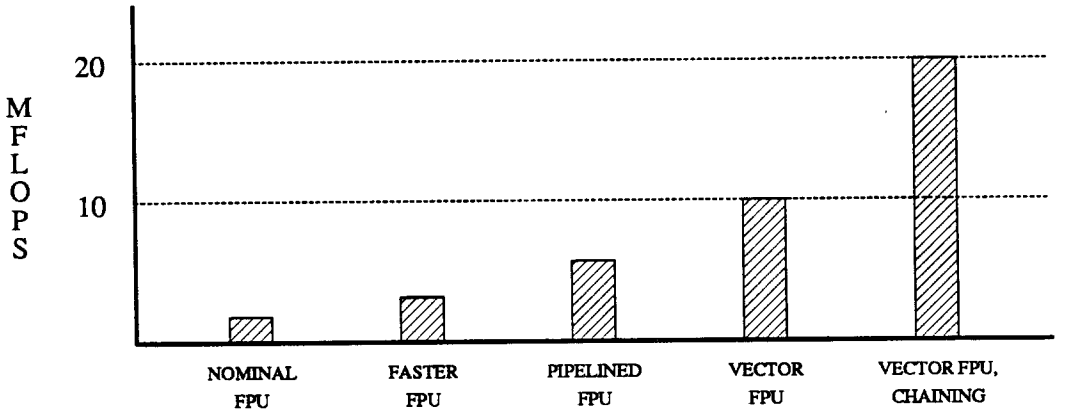


Figure 6-4. Performance of SPUR Architecture for Several FPU Architectures.

This figure shows the *peak* MFLOP rate for various configurations of FPU and vector-FPU hardware. As can be seen, the nominal SPUR configuration can have its performance improved by an order of magnitude with a vector floating-point unit. To prevent stalling of any component, concurrent loads, stores, and vector floating-point unit operations are necessary for the aggressive architectures. The system is substantially more complex, with pipelined function units, vector data cache, and the additional control unit architecture.

6.3.5. Vector Processing Coprocessor Summary

The three areas of focus for high-speed processing are instruction issue rate, function unit execution rate, and memory bandwidth. By trading size for speed, various techniques can be used to enhance each of these. With vector processing, the instruction issue rate is maximized through the use of special, dedicated hardware mechanisms and automatic sequential operand addressing. Function units are segmented into multi-stage pipelines allowing concurrent operation at each stage of the pipe and continuous use. Wide paths and synchronized memories using multiple banks and high-speed buffers combine to facilitate operand transfers between storage and execution elements. All of these interact in many ways. The constraints of the current state of the art in VLSI dictate the temporary boundaries in the design space.

We have shown how the SPUR FPU coprocessor can be extended to effectively provide an order of magnitude increase in performance at a cost of roughly 35% to 50% more chip area in current technology. This will be even more easily realized as technology allows device densities to increase. It is clear that a vector floating-point coprocessor would enable real-time digital signal processing in a conventional workstation, as well as enhance all applications requiring floating-point computation.

With respect to the current coprocessor interface, some changes would be necessary. First, the vector coprocessor needs to manipulate addresses. Since the operations are specialized, and the operands are a specific class, a separate vector data cache is needed to support the type of vector unit described here. A wide path to allow transfer of multiple operands per cycle is imperative for the data flow rates needed to support the computation. It is likely that multiple-port memories couple with

cross-bar mechanisms are needed in the implementation. As well, multi-port register files are needed in the vector unit to allow simultaneous reading for operation and read/load/store for chaining and data transfers. This makes it possible for the vector unit and the CPU to operate totally in parallel with no conflict at the memory.

The instruction issue paradigm from the CPU remains the same, since the vector coprocessor decodes and initiates operations based on the opcode and specifiers it receives. In this sense, the vector unit is closer to a true coprocessor, although it still receives its instructions from the CPU. Exception conditions are handled in a manner similar to the current FPU implementation, with provision for a vector exception state register.

Having briefly explored the realm of high-speed vector floating-point computation, we next consider coprocessors that support execution of high-level languages.

6.4. A Language Coprocessor for SPUR

Along with the increased use of high-level languages often comes the desire to close the *semantic gap* [Myer78], or the difference between the assembly-language level (architecture) of a machine and the high-level languages (HLLs) supported by compilers on the machine. Some researchers contend that if the instruction set more closely matches the semantics of the particular high-level language, the easier it will be for the compiler and ultimately the user to produce efficient code. In some cases, VLSI implementations [Bata82, Boss87] or special micro-coded machines dedicated to a specific language [Moon85] have been designed and built or methods proposed for the process of designing directly interpretable languages [Bose85].

Others believe this process of specialization is self-defeating, particularly when considering the constraints of VLSI implementations and how rapidly the technology is changing. We feel it is better to provide hardware for execution of the most fundamental and frequent operations, and leave infrequent specialized functions to software. Nevertheless, there has always been a desire for some form of architectural support for programming languages.* As was shown in the example of floating-point arithmetic, software implementations can simply be unacceptable, and specialization is needed in the form of a hardware floating-point coprocessor. Is there a case to be made for coprocessors supporting HLLs?

One language that has received considerable attention over the past decade is Prolog, the most popular language for logic programming. Some researchers in artificial intelligence believe that logic programming provides a better way to express problems in machine language than traditional high-level languages do. Many implementations of Prolog have came through refinements of the Warren Abstract Machine (WAM) specification of a Prolog instruction set [Warr83]. Warren's instructions and the Prolog tokens have a close correspondence, making translation a simple process. Thus, most compiled versions of Prolog are based on the WAM definition.

Over the years, several things have been done to improve Prolog run-time performance, including sophisticated compiler techniques, special microcoded devices, and dedicate VLSI processors [Tick83]. The Berkeley Prolog Machine (PLM) [Dobr87] is a TTL implementation of a loosely-coupled microcoded coprocessor in conjunction with a general-purpose host. The PLM machine is capable of executing a set of 14 standard Prolog benchmarks about 10 times faster than a general purpose host (DEC 2060), and is considered one of the world's fastest dedicated implementations for the language.

In [Borr87], a comparison between PLM and a SPUR-based implementation is discussed. Without any special-purpose hardware, a simple macro-expansion of compiled PLM instructions into SPUR code sequences results in performance between 30% and 60% of the PLM. However, the static SPUR code size is about 15 times larger than the PLM code. By incorporating a coprocessor dedicated to implementing the most frequent operations and leaving the rest to standard instructions, along with some minor modifications to the SPUR CPU architecture and floating-point coprocessor interface*, the

*An interesting retrospective on the subject is contained in [Ditz80] with recent conferences devoted to exploring such ideas [ASPL82, ASPL87].

SPUR code size ratio is reduced to 3.5 times that of PLM, with execution time 10% better. The performance improvement comes from the pipelined architecture of SPUR, which takes advantage of the parallelism available at the macro-code level, which PLM does not implement.

One of the main points made in [Borr87] is that a general purpose computer with a tightly-coupled coprocessor for Prolog is particularly useful in mixed-paradigm programming systems, where other languages must be supported besides logic programming. SPUR executes other high-level languages very well, whereas the PLM implementation of mixed-language applications would be much less efficient, and would suffer in performance as a result of the specialization.

In terms of performance, Figure 6-5 illustrates the comparison of several implementations of Prolog. A compiled version of Prolog on a VAX 11/780 runs at 15000 LIPS (logical inferences per second). Using a Symbolics system (a special-purpose computer designed for LISP) an execution rate of 110,000 LIPS is achieved, or an improvement of 7.3 times faster than the VAX 11/780. The SPUR macro-expansion runs at 184,000 LIPS, or about 12.3 times faster than the VAX. Aquarius I runs at 305,000 LIPS (20.3 times faster), Tick and Warren VLSI Prolog machine runs at 415,000 LIPS (27.6 times faster), and the TTL Berkeley PLM plus compiler version runs at 425,000 LIPS (28.3 times faster). Finally, the modified SPUR system with a proposed Prolog coprocessor could run at 465,000 LIPS, about 31 times faster than the VAX implementation.

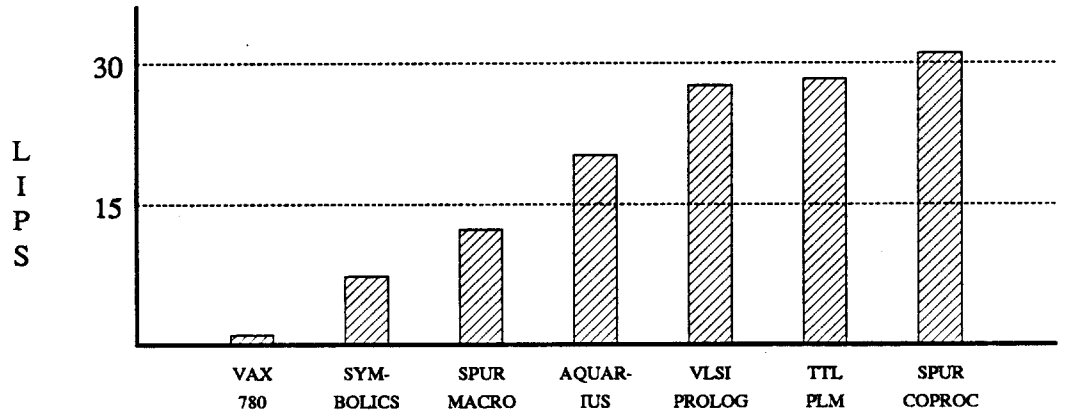


Figure 6-5. Peak LIPS for Several Prolog Architectures.

This figure shows the logical inferences per second for several architectures running Prolog. Using the VAX 11/780 speed as unity, the ratios of speeds are 1.0 to 7.3 to 12.3 to 20.3 to 27.6 to 28.3 to 31 for the architectures studied.

6.5. Chapter Summary

In this chapter, we have considered three applications which benefit from the inclusion of specialized coprocessors to support their functionality. We determined that with a small increase in size and complexity of the current SPUR floating-point coprocessor, more than an order of magnitude performance improvement could easily be achieved with a vector floating-point coprocessor, if a vector data cache is provided. With provision for a dedicated vector-data cache, this provides the performance needed to address the application of digital signal processing in a conventional workstation environment. This general purpose approach provides many advantages, including operating system support,

*To the CPU instruction set is added a *read-and-compare-tag* instruction. To the coprocessor interface is added the cache address bus, cache operation lines, a page fault line, and coprocessor memory access line to arbitrate with the CPU. The Prolog coprocessor is much more general than the SPUR FPU and requires that it be able to manipulate its own data addresses.

generalized compilers and assemblers, data storage and handling, and so forth.

When considering digital signal processing, the vector floating-point capability offers a viable alternative to doing research using high-performance single chip DSPs, which require specialized development environments, program development in low-level assembly languages, and detailed knowledge of the microarchitecture of the device to be able to achieve maximum performance. As well, such systems have limited resources in terms of data access and storage capabilities, pre-defined user interfaces, and debugging tools.

Finally, a coprocessor to support the execution of Prolog was shown to be about equal in performance to highly specialized architectures dedicated solely to the language. Slight modifications to the SPUR instruction set, including a read-and-compare-tag instruction, and the coprocessor interface, including several signals to allow it to manipulate memory independent of the CPU, were necessary to do this.

The flexibility provided by the general purpose system coupled with specialized hardware augmenting certain kernel functions is an extremely effective means of improving computer system performance. It is more cost effective, less prone to obsolescence, and affords a rapid means of evolution for higher performance for present languages and program constructs, and new capabilities in other untried areas.

<This page is intentionally blank.>

7 Discussion, Conclusions, and Future Work

7.1. Introduction and Overview

In this chapter, the basic philosophy that motivated our research is restated. Previous chapters are reviewed and the overall results of the dissertation are summarized and contrasted with other research in this area. The implications of our results, as related to other application-specific integrated circuit implementations or dedicated computational architectures, are presented. To conclude, we suggest areas of future research in computer architecture using coprocessors and software systems needed to support them.

7.2. Philosophy

Uniprocessor computer system performance may be characterized by the following relationship [Henn85]:

$$Time_{progam} = instructions/program * cycles/instruction * time/cycle \quad (7\text{-}1)$$

> The time to execute a program is equal the total number of instructions executed multiplied by the average number of machine cycles per instruction multiplied by the time per machine cycle.

Reduced program execution time results from reducing each of the three terms on the right-hand side of the relationship.

Hardware technologists focus on reducing the third term by providing smaller, ever-faster, and more dense integrated circuits. Software technologists, such as compiler writers and algorithm specialists, focus on reducing the first term by reducing the number of instructions executed by using efficient and clever algorithms.

Computer architects concentrate on the first and middle terms. Even with rapid technological advances, the resources available on a single silicon chip to implement a VLSI architecture are usually

scarce. Consequently, VLSI reduced instruction set computers try to conserve those resources by reducing instruction complexity and then use those same resources to implement the most often-occurring events quickly.

In this dissertation, we explored an extension of the RISC philosophy to components other than the CPU. We called these specialized devices *coprocessors*, and used the term to refer to hardware that is designed to improved performance for any specific task normally done in software. It was our thesis that RISC coprocessors are an effective means to reducing the number of instructions per program and the number of effective cycles per instruction. We focused on one aspect of the design — the coprocessor interface to the rest of the system — in testing the hypothesis. We also considered the application of RISC-like coprocessors to other computationally intensive tasks.

7.3. Research Results

In Chapter 3, we developed a method for evaluating coprocessor effectiveness. We identified and categorized coprocessors according to function, the placement within the system topology, and the fundamental models for control flow and data flow. We identified and quantified factors that are often ignored in terms of their effect on system performance. Based on the premiss that most cycles of many forms of computation are spent in relatively small loops, we developed an analytical model of coprocessor performance that is relevant for a wide range of applications. The model accounts for system influences (such as cache-service time), software influences (such as the interaction between instructions), and hardware influences (such as function unit execution time). The model splits execution time into two components: operation time and overhead time. Instructions executed to perform housekeeping operations, although absolutely necessary, are considered overhead. Coprocessor operations overlapping non-coprocessor operations are used to effectively eliminate overhead time. Using the model, a coprocessor implementation can be characterized in terms of its *effectiveness* and *utilization*:

- effectiveness is a measure of how well the coprocessor performs its function relative to the software it replaces;
- utilization is a figure of merit relating the amount of time a coprocessor is in actual use compared to the total time it could be in use.

The operation and overhead times determine the effectiveness and utilization of a coprocessor.

In Chapter 4, the SPUR floating-point unit, a single-chip CMOS implementation supporting the ANSI/IEEE Standard P754-1985 for binary floating-point arithmetic was analyzed. The performance of this tightly-coupled coprocessor was characterized with respect to its interface to the rest of the SPUR system. We contrasted the performance of the SPUR FPU implementation with commercial floating-point units supporting the standard produced by Intel and Motorola. For each system, we evaluated the effects of:

- bus width between the floating-point unit and operand storage,
- execution time of the fundamental hardware units, and
- influence of cache service time on overall performance.

We observed that commercial implementations tend to balance the amount of time spent in computation with the amount of time spent in overhead. We determined that the commercial systems using a floating-point coprocessor sacrificed much of the advantage of having a second execution unit to overhead factors associated with the following:

- the concurrent execution model supported by the architecture,
- the operand transfer protocol implemented in the hardware,
- the limitations of a narrow data path between the coprocessor and storage,
- the inherent serialization of operations forced by the semantics of certain often-used complex coprocessor instructions, and
- the asynchronous execution model supported by the architecture.

We showed that, in some cases, the mechanisms used to provide CPU/coprocessor concurrency intended to increase utilization actually reduced effectiveness and resulted in longer execution times.

We determined that overhead associated with the concurrent execution model caused the performance degradation.

We found that commercial systems provided more functionality at the hardware level, but did so with multi-cycle, microcoded implementations. We determined that execution times were significantly longer than for a stream-lined RISC floating-point implementation of the four basic arithmetic operations, and overall system performance of the microcoded implementations was between three and 50 times slower.

We determined that the coprocessor interface paradigm used by some commercial implementations did not scale well with increasing floating-point execution unit speed, and that in some cases more than 90% of all execution cycles would be spent on overhead if such architectures were used with state-of-the-art floating-point coprocessors. We concluded that coprocessor interface architectures must change dramatically to keep pace with the rapid advance in CPU execution rates.

In Chapter 5, we analyzed problems associated with path planning to determine if coprocessor technology would be effective in realizing the path optimization function. We used the ideas and analysis tools developed in Chapter 3 and the experience with the detailed floating-point coprocessor implementation of Chapter 4 to guide the research. Shortest-path optimization was selected, since it is often considered the most fundamental and important of all combinatorial optimization problems.

We found no discussion of commercial hardware or coprocessor implementations of the path optimization function in the open literature. Publications discussing path optimization referred only to software implementations using general-purpose computers or special multiprocessor architectures running parallel versions of scan-based algorithms. We found no discussion of sophisticated path planning algorithms, such as Dijkstra's shortest path. In our approach, we developed and analyzed several algorithms to determine those with the best performance characteristics, and then determined which algorithms provided the most suitable model for coprocessor implementation. To do this, we developed two simulators to model and quantify various performance characteristics of path optimization algorithms. We determined that:

- scan-based techniques are algorithm dependent and vary more than a factor of five in performance for the same data input;
- scan-based techniques are data dependent and vary more than an order of magnitude over various data sets;
- scan-based techniques universally converge much faster with coherent data (i.e., from well-behaved data models or realistic terrain-like maps);
- Dijkstra's shortest path is deterministic across all data sets and between two and nearly 200 times faster than scan-based techniques for the same data;
- minimization comparisons must include diagonal near-neighbors to produce optimum shortest paths, even with coherent data; and
- total cost map entries can be limited to 16-bit quantities with provision for overflow, reducing overall memory requirements.

We determined that our uniprocessor software implementation of path optimization using a simple version of Dijkstra's shortest path algorithm was about twice as fast as a 40-node multiprocessor implementation of a scan-based algorithm and more than an order of magnitude better than previous uniprocessor results reported in the literature [Lind86]. From the algorithm analysis, we presented several architectures for implementing path optimization, including special purpose pipelined and parallel hardware.

We outlined a RISC architecture for a scan-based path optimization coprocessor based on the SPUR system model, and discussed various optimizations at the microarchitecture level to achieve real-time path optimization performance. Our path optimization coprocessor is roughly able to meet real-time performance requirements at a substantially lower cost and complexity than other architectures capable of the same or better performance. We concluded that path optimization is a problem suitable for implementation using coprocessor technology and that a RISC coprocessor based on the SPUR model works reasonably well.

In Chapter 6, we briefly explored extensions to the SPUR coprocessor model in areas of digital signal processing, vector floating-point arithmetic, and Prolog language implementation.

For signal processing, we determined that a pipelined version of the SPUR FPU could provide near-real-time performance in a workstation environment. No other changes to the architecture would be necessary. We showed that the microarchitecture of the FPU could be enhanced with multiple function units to achieve performance within a factor of two to five of state-of-the-art DSP chips. Although this performance is slightly inadequate for real-time processing, we determined that such a capability in a general-purpose workstation would provide a rich and easy-to-use research environment.

For vector floating-point arithmetic, we showed that the SPUR system would have to change significantly, providing a separate vector data cache, a vector control and address unit, the addition of vector instructions, and mechanisms to retain state and exception-condition information. Nevertheless, the fundamental model provided by the SPUR system — decoupled execution, load/store, and so forth — accommodated the needs of a vector coprocessor and presented an attractive area for further research.

For Prolog language support, research completed at Berkeley by [Borr87] showed that minor instruction set changes and the ability to manipulate addresses were needed to efficiently implement a Prolog coprocessor using the SPUR model. The Prolog coprocessor instruction set included a few of the most frequently occurring operations in the WAM Prolog definition and left other operations to SPUR code. The SPUR Prolog coprocessor system would be slightly faster than some highly specialized architectures reported in [Dobr87]. We concluded that a RISC approach to supporting the Prolog language using a coprocessor to speedup the most frequently occurring operations was feasible.

7.4. Summary and Future Work

It was our thesis at the beginning of this research that RISC CPU architectures coupled with RISC-like coprocessors could provide better performance improvements for many software tasks that are currently provided by CISC CPU's and their companion CISC-like coprocessors. Also, that the decoupling between the CPU and coprocessor afforded by RISC architectures allows maximum utilization of special purpose devices in general purpose computer systems. We are encouraged by the results reported here and believe that the ideas hold true for a broad range of applications.

To generalize the experience gained and reported in this dissertation, we briefly review our method to determine if a particular application would benefit from a special purpose hardware coprocessor:

- Conduct an extensive algorithm analysis to determine the requirements of the processing function;
- Identify fundamental operations and the corresponding data structures needed for each of the algorithms;
- Identify operations that can be done in parallel, those that occur most frequently, and those that require flexibility;
- Consider alternative implementations for each function or sub-function, from full software to full hardware;
- Predict or simulate incremental and overall system performance improvement for each alternative implementation;
- Reiterate the preceding steps until a satisfactory solution is chosen.

Future work in the area of coprocessor architecture could involve:

- extensions to the current SPUR coprocessor interface model,
- use of multiple homogeneous coprocessors on specific applications, or
- development and use of new RISC-like coprocessors.

Extending the current implementation. The coprocessor interface is strongly influenced by the needs and requirements of the floating-point coprocessor for the SPUR processor node. To make the

interface more general and yet not sacrifice the simplicity of the design, several things might be considered for inclusion in a second version:

- include general-purpose coprocessor instructions in the instruction set architecture;

  The current model for the SPUR instruction set architecture presumes only one coprocessor exists, and includes explicit instructions for the floating-point unit. This dissertation has discussed other applications that would benefit from using coprocessors. The load/store architecture of the SPUR system makes it simple to provide a generalized load or store along with coprocessor identification bits. As well, a generalized set of instructions to initiate actions that must stall if the coprocessor is busy and other instructions that can operate concurrently are needed to facilitate other applications.

- include coprocessor ID bits in the instruction set and coprocessor status bits in the CPU status word;

  Along with the generalized coprocessor instructions, a means to distinguish between homogeneous and heterogeneous coprocessors is needed. The current architecture presumes only one coprocessor exists (the floating-point unit) for simplicity in the prototype design. There is no fundamental reason to be limited to one.

- include multiple exception-condition, test-condition, and status inputs to the CPU;

  A means of signaling exception and normal run-time conditions on the coprocessor is critical to providing a responsive system. Explicit lines should be dedicated to those functions that occur frequently (such as condition test), while less frequent events, such as exceptional conditions, may be encoded if necessary due to pin limitations.

- include the cache address bus and status lines in the interface.

  As determined in several of the examples studied in earlier chapters of this dissertation, some functions must manipulate address and load and store operands independent of the CPU to be effective. The ability to control the address bus imposes some amount of arbitration and synchronization between the coprocessor and the CPU, but there is no reason to believe that system performance would suffer as a result.

Use of multiple, homogeneous coprocessors. For many problems, a SPUR CPU equipped with multiple FPU's may prove to be an effective way to achieve the advantages of pipelining without having pipelined floating-point execution units. The extent to which multiple FPU's could be useful depends on the latency of the function units and the series of computations performed. The trade-offs between these factors should be studied. The use of a multiprocessor path optimization system would theoretically achieve linear speedup with the coprocessor architecture presented in Chapter 5. However, the effects of multiple processor access to shared data and the incumbent memory-latency degradation associated with caching behavior should be determined before projecting the speedup and other advantages of such a system.

In our work reported in [Bose88a, Bose88c], we have considered some of the issues related to the SPUR multiprocessor workstation running scientific programs and the effects of contention for shared memory. More research is needed to determine the proper balance between coprocessor speed and the demands it places on other system resources.

Developing new coprocessors. First, for distributed and networked computer environments, some researchers have suggested that certain aspects of low-level communication handling protocols would benefit greatly from specialized hardware [Ches88].

Second, researchers are typically unable to obtain good performance evaluations of existing or new hardware. Various techniques including modifications of microcode, retro-fitting existing machines with special purpose hardware, or simply adding code to applications have been used. We believe a performance monitor coprocessor based on the SPUR interface model could provide a much needed yet simple tool for computer architects and performance analysts to evaluate their work.

Third, building on the analysis and evaluation methods reported here and using circuit specification and synthesis tools, such as reported in [Bori88], rapid evaluation and specification of coprocessor architectures and interfaces would seem to be a natural adjunct to the logic synthesis and

silicon compiler research being pursued at a number of research institutions. We believe that the SPUR approach of providing a tightly-coupled coprocessor interface tied to the CPU pipeline and yet decoupled in terms of operation and operand transfers provides many advantages and yet is simple enough to be formalized for ASIC designs using synthesis tools.

Finally, the question of whether increasing density in silicon technology should be used to place coprocessor functions "on-chip," or whether there are still advantages to having separate devices needs to be considered. Such things as specialized data and specialized operations on those data must be considered. The answer to these and many other related questions that arise can only be found in the context of an application, and we look forward to pursuing several of them in the years to come.

A The SPUR Floating-point Coprocessor Interface Description

This appendix is adapted from [Hans86] and briefly describes the SPUR coprocessor interface in the context of the floating-point unit. The interface provides enhanced performance potential by allowing parallel operations between the SPUR processor and SPUR coprocessors. A decoupled control and execution architecture allow data transfers to proceed while coprocessor functions are performed. Implicit and explicit synchronization mechanisms provide the programmer complete control and flexibility. On-chip coprocessor register files and a wide data path between the memory and coprocessor minimize data transfer overhead. An intelligent interface control unit provides parallel decoding of instructions for maximum performance. Other coprocessor functions applicable to dynamic programming optimization, signal processing, image processing, performance monitoring, language coprocessors, workstation graphics, and so forth are being considered, but will not be reported here.

A.1. Introduction

The SPUR CPU is a custom VLSI-32 bit general-purpose host targeted to support Lisp and other high-level language software environments. The RISC-like architecture provides high performance for a wide range of applications.

Traditional von Neumann computer architectures have achieved enhanced performance by adding optional hardware to perform tasks that are usually executed in software. These devices are often called *coprocessors* and include attached processors, array processors, floating-point accelerators, data channels, graphics display processors, performance monitors, and so forth. Thus, a coprocessor is an optional piece of hardware that replaces a piece of software for a higher level of performance.

Many peripheral devices as well as more closely coupled coprocessors fall in this general category. It is nevertheless important to recognize a distinction between standard peripheral hardware devices and *tightly coupled* coprocessors: the programming model for the coprocessor differs from that of peripheral devices. Standard peripheral hardware usually appears to the programmer as a set of registers in the memory space of the main processor. The programmer must consider the communication protocol and implement the interface between the peripheral and the device in software.

In contrast to this, the tightly coupled coprocessor adds special instructions to the CPU instruction set that allow the programmer to utilize the coprocessor capabilities. It may also provide additional registers and data types that are not directly supported by the main processor architecture. However, certain interactions needed between the main processor and the coprocessor (i.e., the communications protocol) are implemented in hardware and are transparent to the programmer. Thus, the coprocessor can extend the functions provided to the user without appearing as hardware external to the main processor. This provides a more uniform programming model from a user point of view.

The SPUR system employs an optional special purpose device for floating-point arithmetic. This device will support the IEEE Standard P754 for add, subtract, multiply and divide in single, double, and extended precisions. (Other functions, such as transcendentals, are handled by runtime routines.) We refer to this as the SPUR Floating-point Unit, or simply FPU. For documentation purposes, it would seem logical to refer to all signals and mnemonics related to the coprocessor to be designated "CP". Other applications are being considered besides floating-point arithmetic, but this report will focus on the FPU. Thus, to avoid confusion between the CPU and CP designations, the coprocessor interface signals, blocks, modules and functions will be designated with the "fpu" prefix. The first generation SPUR system will support a floating-point coprocessor (FPU) and a performance monitor (PMC) [Fauc86]. Later generations will consider other applications.

Section 2 of this report provides a brief overview of the SPUR coprocessor interface and functions. Section 3 provides a greater degree of detail and timing diagrams for various operations, instructions, and the interaction between the CPU and FPU.

A.2. Floating-point Coprocessor Interface Overview

From the assembly language programmers point of view, the SPUR FPU has 15 read/write 87-bit operand registers and one read/write control/status register (nominally 64 bits). The register bits are defined from left to right (with MSB at left-hand side) as follows:

# of BITS	1	17	64	2	3
FIELD	SIGN	EXPONENT	FRACTION	ROUND TAG	DATA TYPE

The FPU is a load/store architecture. Consequently, all arithmetic operations involve three registers: two source and one destination.

A.2.1. Instructions

There are 18 operations defined for floating-point arithmetic and general coprocessor functions (load, store, etc) in the SPUR instruction set. They are listed in Table A-1. It should be noted that all LOAD instructions have two forms, depending on the cache operation involved: simple read or read with ownership. For example, the two opcodes for loading single precision operands are LD_SGL and LD_SGL_RO.

Table A-1. SPUR Floating-point Unit Instructions

Arithmetic Operations, Operand Conversion, and Compare				
Instruction Syntax		Instruction Semantics		
FADD	Rd,Rs1,Rs2	FPU Rd	⇐	FPU Rs1 + FPU Rs2
FSUB	Rd,Rs1,Rs2	FPU Rd	⇐	FPU Rs1 - FPU Rs2
FMUL	Rd,Rs1,Rs2	FPU Rd	⇐	FPU Rs1 * FPU Rs2
FDIV	Rd,Rs1,Rs2	FPU Rd	⇐	FPU Rs1 / FPU Rs2
FABS	Rd,Rs1,0	FPU Rd	⇐	FPU Rs1 with sign = 0
FNEG	Rd,Rs1,0	FPU Rd	⇐	FPU Rs1 with inverted sign
FCMP	cond,Rs1,Rs2	FPSW(cond)	⇐	result
CVTS	Rd,Rs1,0	FPU Rd	⇐	(convert to single) FPU Rs1
CVTD	Rd,Rs1,0	FPU Rd	⇐	(convert to double) FPU Rs1
Load FPU Registers and FMOV				
Instruction Syntax		Instruction Semantics		
LD_SGL,RO	Rd,Rs1,RC	FPU Rd	⇐	M [(Rs1 + RC)
LD_DBL,RO	Rd,Rs1,RC	FPU Rd	⇐	M [(Rs1 + RC)
LD_EXT1,RO	Rd,Rs1,RC	FPU Rd	⇐	M [(Rs1 + RC)
LD_EXT2,RO	Rd,Rs1,RC	FPU Rd	⇐	M [(Rs1 + RC)
FMOV	Rd,Rs1,0	FPU Rd	⇐	FPU Rs1
Store FPU Registers				
Instruction Syntax		Instruction Semantics		
ST_SGL	Rs2,Rs1,SC	FPU Rs2	⇒	M [(Rs1 + SC)
ST_DBL	Rs2,Rs1,SC	FPU Rs2	⇒	M [(Rs1 + SC)
ST_EXT1	Rs2,Rs1,SC	FPU Rs2	⇒	M [(Rs1 + SC)
ST_EXT2	Rs2,Rs1,SC	FPU Rs2	⇒	M [(Rs1 + SC)

A.2.2. Control Flow

The FPU coprocessor has two major functional units: the interface control unit (ICU) and the execution unit (EU). The clocking scheme of the FPU is identical to the CPU: 4 non-overlapping phases per cycle. (Refer to Section 3 for more details.) In phase 3 (phi3) of every cycle, the FPU ICU accepts and decodes the INSTRUCTION BUS fragment issued on *fpuOPCODE_CV3* lines, and initiates operation in the subsequent cycle if it is an FPU operation. This continues until cycle N (N is the number of execution cycles for the particular instruction being executed). The *fpuBusy_C4* signal is disasserted in the last execution cycle (register write) to signal when the FPU EU is done. (See Section 2.6 for complete definitions of signals.)

Under normal circumstances, CPU and FPU instructions execute in parallel. This parallelism is controlled in two possible ways: (1) explicit: the *fpuParallel* bit in the Upsw (user process status word in the CPU) may be set, which will allow overlap of CPU and FPU operation instructions, and (2) implicit: the assertion of the *fpuBusy_C4* line will prevent the CPU from issuing FPU operation instructions if the FPU is still in the execution phase of a previously issued instruction (as signaled by *fpuBusy_C4*). When overlap is prevented, the CPU always stalls until the *fpuBusy_C4* line is not asserted.

A.2.3. Data Flow

Data flow between the FPU and the SPUR data cache memory is directly controlled by the CPU. The data path to the cache is 64 bits wide. Double precision operands are loaded in one cycle. Loads may proceed in parallel with FPU operation, since the FPU register file is dual ported. The FPU pipeline is similar to the CPU pipeline: an FPU load requires the instruction fetch, effective addresses calculation, memory access, and register write cycles. Since there is no operand forwarding in the FPU, the load target is not ready for use in the FPU until the third instruction following the load instruction.

A.2.4. Performance

Table A-2 lists the execution cycles needed to complete FPU arithmetic operations. Loads and stores are considered single-cycle operations, and are discussed in Section 3.2.1.

Table A-2. SPUR FPU Execution Cycles for Arithmetic Operations

Instruction		Cycles (operation only)
FADD	Rd,Rs1,Rs2	3
FSUB	Rd,Rs1,Rs2	3
FMUL	Rd,Rs1,Rs2	8
FDIV	Rd,Rs1,Rs2	19
FABS	Rd,Rs1,0	3
FNEG	Rd,Rs1,0	3
FCMP	cond,Rs1,Rs2	3
CVTS	Rd,Rs1,0	3
CVTD	Rd,Rs1,0	3

Studies comparing the SPUR FPU with commercial microprocessor-based systems employing VLSI floating-point coprocessors indicate that the SPUR-FPU combination can execute several floating-point intensive benchmarks between 3 and 50 times faster than other systems [Hans88]. The main performance advantages come from:

- the dual ported register file allowing data loads and stores during FPU operation,
- efficient mechanisms for synchronizing parallel execution of the FPU and CPU,
- the wide data path between the FPU and cache memory, and
- very efficient algorithms and hardware structures for the four operations implemented on-chip: add, subtract, multiply, and divide.

A.2.5. Programming Interface

The FPU effectively adds new data types, new registers, and new instructions to the CPU. The coordination of the processor-coprocessor operation is handled mostly by the programming languages and coprocessor interface automatically. The FPU architecture is Load/Store, with arithmetic operations between FPU registers. The hardware is invoked directly by program instructions, and no recompilation is necessary for systems that are not equipped with an FPU. Simple link-time command arguments direct the loading of algebraic routines in the absence of the FPU. One bit in the Upsw causes the CPU to trap to algebraic routines when FPU instructions are encountered and an FPU is not available in the system.

A.2.6. Hardware Interface

Figure A-1 shows how the FPU is connected as a coprocessor in the SPUR system. Figure A-2 shows the logical interconnections between the CPU and FPU. The CPU and FPU both use a 4-phase non-overlapped clocking scheme as illustrated in Figure A-3. The coprocessor interface signals fall into three groups:

- instruction: opcode and FPU register specifiers,
- control (to FPU): new instruction valid, suspend FPU operation, and
- status (from FPU): FPU busy, FPU exception, FPU compare result.

A.2.6.1. CPU to FPU Signals

Below is a brief description of the signals from the CPU to the FPU.* For a more detailed discussion, please refer to [Hans86].

fpuOPCODE_CV3: 7 bits. This specifies the opcode of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.
fpuRS1_CV3: 5 bits. This specifies the first source register of the instruction, which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.
fpuRS2_CV3: 5 bits. This specifies the second source register of the instruction, which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.
fpuRD_CV3: 5 bits. This specifies the Destination register of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.
fpuNewInstr_CV3: 1 bit. Asserted by the CPU whenever a valid instruction of any variety is issued. The CPU starts driving this signal at the beginning of phi3.
fpuSuspend_CV4: 1 bit. Asserted by the CPU during any pipeline suspension, except when the pipeline is suspended due to *fpuBusy_C4*. (This effectively is a signal to stall FPU register writes.) The CPU starts driving this signal at the beginning of phi4.

A.2.6.2. FPU to CPU Signals

The signals between the FPU and CPU which provide status are described next. Many of the details of operation are contained in Section 3 of this report: Coprocessor Interface Details. Note: If the FPU begins driving a signal at the beginning of phiN, it is assumed that the signal is stable and latchable at the end of phiN on the CPU.

fpuBusy_C4: 1 bit. Asserted by the active coprocessor (the FPU) to indicate that it is busy. The FPU starts driving this signal at the beginning of phi4. The CPU latches it at the end of phi1.
fpuExcept_C4: 1 bit. Notifies the CPU that an error condition exists in the coprocessor. The FPU starts driving this signal at the beginning of phi4. The CPU latches it at the end of phi1.

*If either the CPU or the FPU begins driving a signal at the beginning of phiN, it is assumed that the signal is stable and latchable at the end of phiN at the destination.

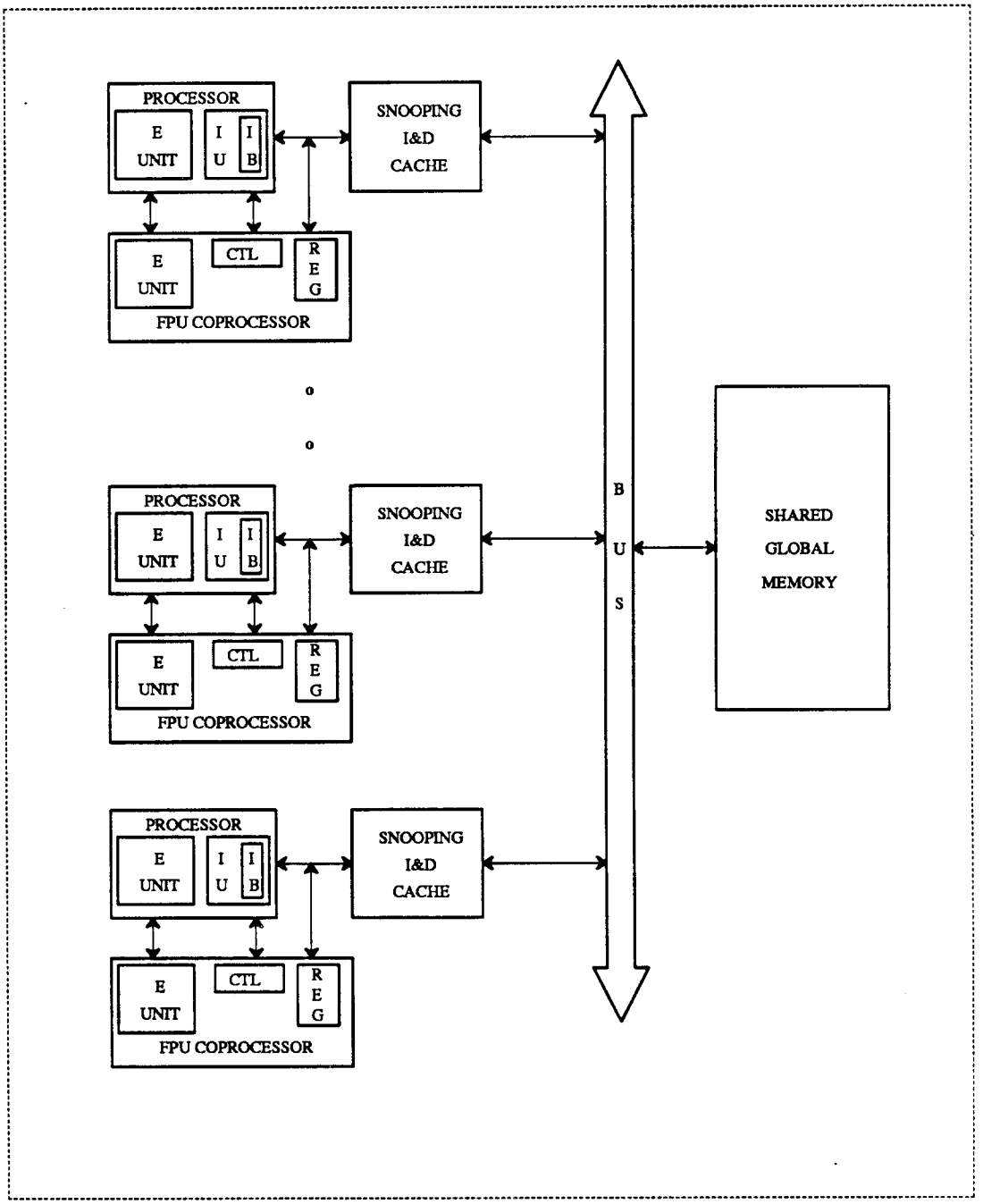


Figure A-1. The UC Berkeley SPUR Multiprocessor System.

The processor is a single chip with an on-board instruction buffer (IB). The floating-point unit (FPU) is tightly coupled to the CPU via the local processor bus. The cache controller is integrated on one chip with off chip tag and data RAMs. The caches work together to implement a cache consistency protocol, described in [Katz85]. Shared memory and I/O devices are accessible through the system bus.

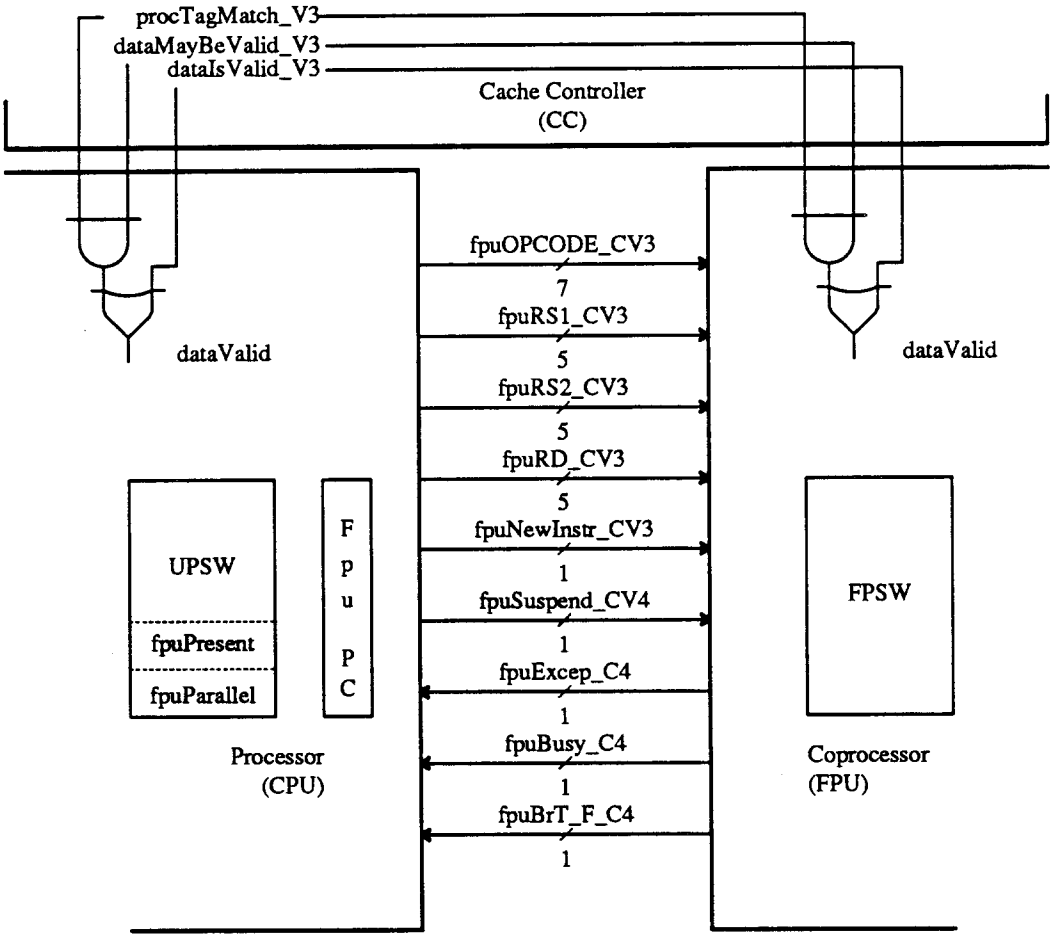


Figure A-2. The SPUR Floating-point Coprocessor Interface.

This figures shows the interconnections between the SPUR CPU and FPU. Signals *dataMayBeValid_V3*, *tagMatch_V3*, and *dataIsValid_V3* come from the cache controller to both the CPU and FPU.

fpuBrT_F_C4: 1 bit. A signal coming from the FPU Fpsw which indicates the result of the last FCMP instruction. The FPU starts driving this signal at the beginning of phi4. The CPU latches it at the end of phi1.

Table A-3 summarizes the signals between the CPU and FPU and indicates the phase in which the signals change (driven either by the CPU or FPU) and are latched (by either the CPU or FPU).

A.2.6.3. CPU UPSW and FPU PC Registers

Besides the signals above, the CPU must maintain the following information in the Upsw to support coprocessors:

fpuParallel: 1 bit. When asserted, it enables parallel operation of the CPU and the FPU. If this bit is not set, parallel operation is prohibited (forcing sequential mode). Parallel operation is described later.

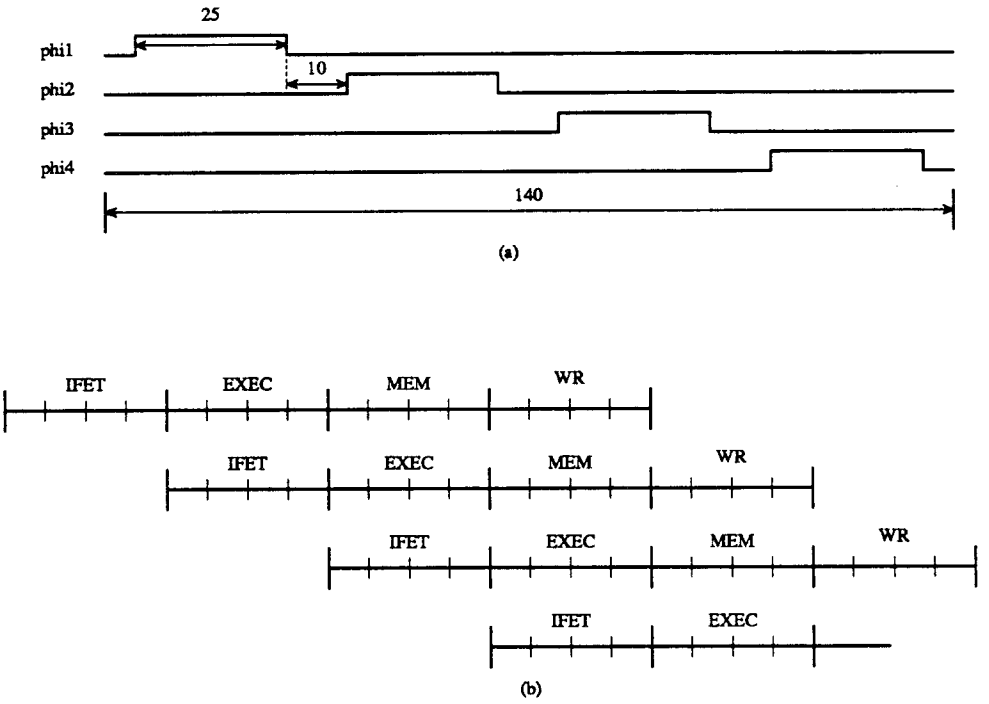


Figure A-3. CPU and FPU Pipeline Stages.

As shown in Par(a), clocking is 4-phase, non-overlapping, with 25 nsec high levels, 10 nsec underlap low levels yielding a 140 nsec total cycle time. In Part (b), register operations do not use the Mem cycle, permitting instruction prefetching or operand access during those times. Although it is theoretically possible to issue one instruction per cycle (as shown above), cache misses and/or a busy coprocessor (cannot begin execution of a second instruction until the first is done) will increase the number of effective cycles per instruction completed to more than 1.0. Simulations indicate that between 1.6 and 2.0 clock cycles per instruction are necessary.

fpuEnable: 1 bit. When asserted, it indicates to the CPU that an FPU device is available in the system. When not asserted, the CPU traps to runtime routines to emulate floating-point hardware operations.

The parallel execution of CPU and FPU instructions requires that the CPU maintain a copy of the last FPU instruction's address which is needed in the event of an exception. Exception handling routines must determine what action is necessary based on the instruction that faulted. However, with parallel operation between the CPU and FPU, the program counter inside the CPU may not be pointing at the FPU instruction that causes the FPU exception. This implies that the CPU must have a special register that stores the address of the last FPU instruction the CPU issued. This special register is the *FpuPC*, which is loaded for all FPU operations.

A.2.7. Floating-point Unit Micro-Architecture

The description of the internal architecture and structure of the floating-point unit is beyond the scope of this report, and is discussed in [Bose88]. The concluding section of this appendix briefly describes some of the additional information contained in [Hans86] pertaining to the SPUR floating-point coprocessor interface.

Table A-3. SPUR Floating-point Coprocessor Interface Signals and Timing

	CLOCK EDGE of CYCLE							
SIGNAL	phi1:LE	phi1:TE	phi2:LE	phi2:TE	phi3:LE	phi3:TE	phi4:LE	phi4:TE
fpuOPCODE_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuRS1_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuRS2_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuRD_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuNewInstr_CV3	- -	- -	- -	- -	C(cpu) -	S L(fpu)	- -	- -
fpuSuspend_CV4	- -	- -	- -	- -	- -	- -	C(cpu) -	S L(fpu)
fpuBusy_C4	- -	- L(cpu)	- -	- -	- -	- -	C(fpu) -	S -
fpuExcept_C4	- -	- L(cpu)	- -	- -	- -	- -	C(fpu) -	S -
fpuBrT_F_C4	- -	- L(cpu)	- -	- -	- -	- -	C(fpu) -	S -
dataMayBeValid_V3 (pulse)	- - -	- - -	- - -	- - -	C(cc) - -	S L(cpu) L(fpu)	- - -	- - -
dataIsValid_V3 (pulse)	- - -	- - -	- - -	- - -	C(cc) - -	S L(cpu) L(fpu)	- - -	- - -
procTagMatch_V3 (pulse)	- - -	- - -	- - -	- - -	C(cc) - -	S L(cpu) L(fpu)	- - -	- - -
interrupts	-	L(cpu)	-	-	-	-	-	-

All signals are levels, except as indicated.
All signals shown "-" are assumed stable or unasserted at those times.

phiN	clock phase N.
LE	leading edge.
TE	trailing edge.
_L	signal is asserted low. All other signals are asserted high by default.
_CN	signal changes only during phi N, where N = 1, 2, 3, or 4.
_VN	signal is valid only during phi N, where N = 1, 2, 3, or 4.
_CVN	signal changes only during phi N and is valid by the end of phi N, where N = 1, 2, 3, or 4.
_N	signal has valid and non-zero value only during phi N, where N = 1, 2, 3, or 4.
cpu	SPUR central processing unit
fpu	SPUR floating-point unit
cc	SPUR cache control unit
C(xxx)	the phase-edge where the sender (CPU or FPU) begins CHANGING the signal.
S	the phase-edge where the signal is first assumed STABLE.
L(xxx)	the phase-edge where the receiver (CPU or FPU) LATCHES the signal.

A.3. Overview of Coprocessor Interface Details

The full technical report [Hans86] describes in detail the interaction between the CPU and FPU during normal processing, exception and interrupt handling, and parallel operation. Detailed timing diagrams show the interaction between the CPU and FPU for different instruction sequences and the mechanisms that allow parallel operation, including suspension of the CPU pipeline when the FPU is busy.

There are many special-case situations that must be handled by the coprocessor interface. The report includes in details about:

- the effects of cache misses on the CPU and FPU pipelines,
- CPU or FPU loads followed by CPU or FPU loads with and without cache misses,
- what happens when a cache miss results in a page fault, and
- various exception handling mechanisms, including exceptions during cache misses and subsequent faults.

For example, Figure A-4 shows the trap timing for the CPU, FPU, and memory when a memory reference to a floating-point operand results page or bus fault.

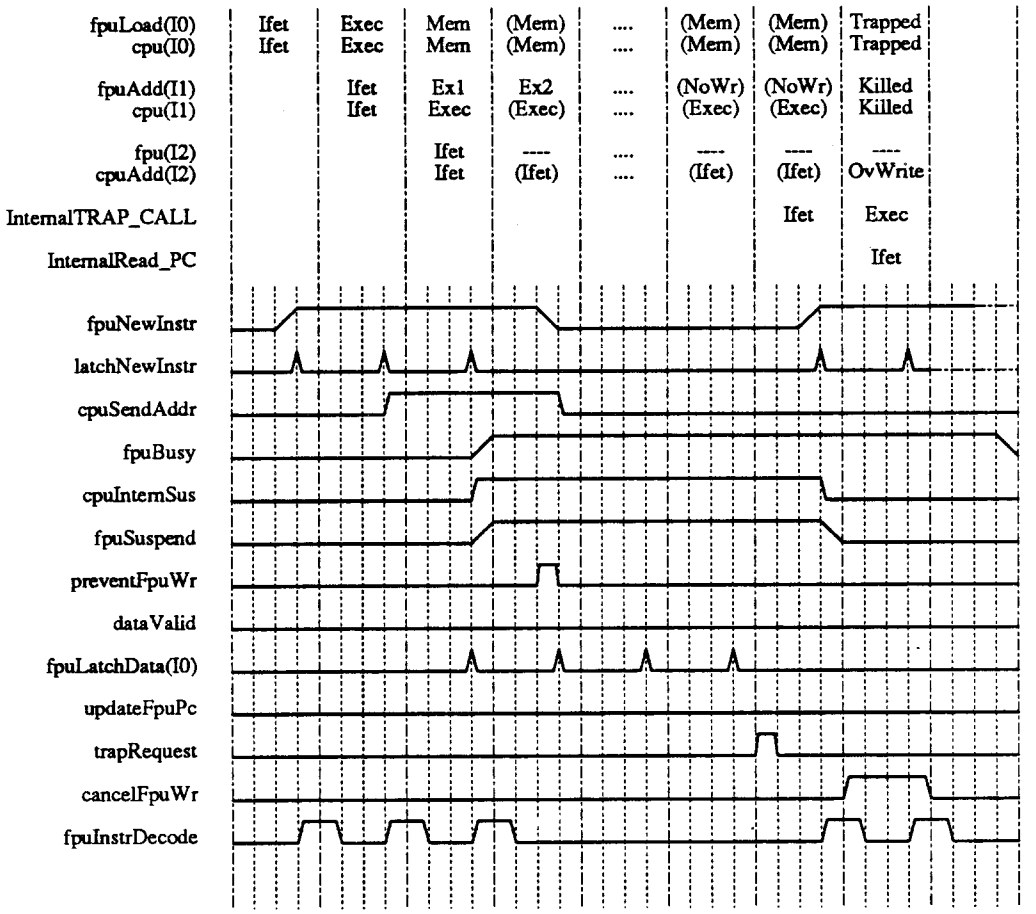


Figure A-4. Trap Timing for FPU Page or Bus Fault.

B FPU Simulation Results, Benchmark Listings, and Commercial Floating-point Arithmetic Coprocessor Instruction Timings

This appendix consists of four sections. Section B.1 includes the C-language version of the microbenchmarks discussed in Chapter 4. Section B.2 includes the SPUR assembly language versions for the microbenchmarks in the Endot *metaMicro* generalized microassembler syntax. Section B.3 includes several tables listing the floating-point performance results for the microbenchmarks using the models of the Intel i80286/i80287, Motorola MC68020/MC68881, and SPUR systems. Section B.4 includes the instruction timings for the Intel i8087 and i80287 Numeric Data Processors and the Motorola MC68881 Floating-point Coprocessor described in this dissertation.

B.1. High-level Language Code Listings of Floating-point Microbenchmarks

The code listings here are in the C programming language, and illustrate different ways that each program can be coded. These, coupled with the various compilers and optimization options, were used to guide our selection of what were representative implementations for each of the functions.

B.1.1. Gaussian Elimination

```
static double x[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
static double y[10] = {.97,.87,.77,.67,.57,.47,.37,.27,.17,.07};
main()
{
  register int i;
  register int count=0;
  double k;
  k = 0.987654321;

  do {
    for(i=0;i<10;i++)
        x[i] = x[i] + (k * y[i]);
  } while (count++ < 5000) ;
}
```

```
static double x[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
static double y[10] = {.97,.87,.77,.67,.57,.47,.37,.27,.17,.07};
main()
{
  register int i;
  register int count=0;
  double k;
  k = 0.987654321;

  do {
    for(i=0;i<10;i++)
        x[i] += ( k * y[i] );
  } while (count++ < 5000) ;
}
```

```
static double x[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
static double y[10] = {.97,.87,.77,.67,.57,.47,.37,.27,.17,.07};
main()
{
  register int count=0;
  register double *px, *py;
  double k;
  k = 0.987654321;

  do {
    for(px = x, py = y; py < &(y[10]); )
        *px++ += (k * *py++);
  } while (count++ < 5000) ;
}
```

B.1.2. Dot Product

```
static double x[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
static double y[10] = {.97,.87,.77,.67,.57,.47,.37,.27,.17,.07};
main()
{
  register int i;
  register int count=0;
  double p=.987654321;

  do {
    for(i=0;i<10;i++)
      p = p + (x[i] * y[i]);
  } while (count++ < 5000) ;
}
```

```
static double x[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
static double y[10] = {.97,.87,.77,.67,.57,.47,.37,.27,.17,.07};
main()
{
  register int i=0;
  register int count=0;
  double p=.987654321;

  do {
    for(i=0;i<10;i++)
      p += (x[i] * y[i]);
  } while (count++ < 5000);
}
```

```
static double x[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
static double y[10] = {.97,.87,.77,.67,.57,.47,.37,.27,.17,.07};
main()
{
  register int count=0;
  register double *px, *py;
  double p=.987654321;

  do {
    for(px = x, py = y; py < &(y[10]); )
      p += (*px++ * *py++);
  } while (count++ < 5000);
}
```

B.1.3. Polynomial Evaluation

```
static double c[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
main()
{
  register int i;
  register int count=0;
  register double k=.987654321;
  register double p=.123456789;

  do {
    for(i=0;i<10;i++)
      p = ( p * k ) + c[i];
  } while (count++ < 5000) ;
}
```

```
static double c[10] = {.93,.83,.73,.63,.53,.43,.33,.23,.13,.03};
main()
{
  register int i;
  register double *pc;
  register int count=0;
  register double k=.987654321;
  register double p=.123456789;

  do {
    for(pc = c; pc < &(c[10]); )
      p = (p * k) + *pc++;
  } while (count++ < 5000) ;
}
```

B.2. SPUR Assembly Language Code Listings of Floating-point Microbenchmarks

The code listings here are in the Endot *metaMicro* generalized microassembler syntax [Ordy83], and include the COMP and ASSM versions discussed in Chapter 4. The listings for Intel and Motorola are not included here.

B.2.1. Gaussian Elimination

```
!    gaussian elimination - for matrix inversion - (fortran version)
!
!    x[i] = x[i] + (k * y[i])
!
!    note: assume word addressed machine; ie, dbl precision operands
!    reqire index to be shifted by 1 to make offset
!

include spurarch.def $
begin
    . = 0 $

START:  sub(r0,r0,r0)       !clear r0 index
        sll(r3,r0,1)        !make offset for c array
        add(r10,r0,8)       !set loopcount
```

```
LOOP:   ld_dbl_r(f2,r3,YARRAY) !load f2 with y data
        sll(r2,r0,1)           !make offset for x array (fpunoop)
        fpunoop                !wait for load y data to complete on fpu
        fmul(f3,f4,f2)         !multiply k (in f4) * y (in f2) leave res in f3
        ld_dbl_r(f1,r2,XARRAY) !load f1 with x data
        sll(r2,r0,1)           !make offset for x array (fpunoop)
        add(r4,r2,XARRAY)      !make store address for x in r4
        fadd(f1,f1,f3)         !x + k*y ==> x
        cvtd(f1,f1,0)          !convert x to dbl prec
        add(r0,r0,1)           !increment index
        sll(r3,r0,1)           !make offset for y array
        st_dbl(f1,r4,0)        !x ==> memory[r4]
        cmp_br_delayed(cc_ne,r0,r10,LOOP) !test index against loopcount

HALT:   sub(r31,r31,r31)       !use ref to r31 to stop simulation
        jump_reg(r0,HALT) !infinite loop

        . = 200 $

XARRAY: constant(100)
        constant(200)
        . = 400 $

YARRAY: constant(300)
        constant(400)
end
```

```
!   gaussian elimination - for matrix inversion - (hand version)
!
!   x[i] = x[i] + (k * y[i])
!
!   note#1: assume word addressed machine; ie, dbl precision operands
!   reqire index to be shifted by 1 to make offset
!
!   note#2: assume the constant k is on the fpu to start with
!

include spurarch.def $
begin
    . = 0 $

START:  sub(r0,r0,r0)          !clear r0 index
        add(r1,r0,r0)          !make offset for x and y arrays
        add(r10,r0,8)          !set loopcount
        ld_dbl_r(f2,r1,YARRAY) !load f2 with initial y data

LOOP:   fmul(f4,f3,f2)         !multiply k (in f3) * y (in f2) leave res in f4
        ld_dbl_r(f1,r1,XARRAY) !load f1 with x data
        add(r4,r1,XARRAY !make store address for x in r4
        add(r1,r1,2)           !make new offset for x and y array
        fadd(f1,f1,f4)         !x + k*y ==> x
        cvtd(f5,f1,0)          !convert x to dbl prec
        ld_dbl_r(f2,r1,YARRAY) !load f2 with next y data
        add(r0,r0,1)           !increment index/counter
        st_dbl(f5,r4,0)        !x ==> memory[r4]
        cmp_br_delayed(cc_ne,r0,r10,LOOP) !test index against loopcount
```

```
HALT:   sub(r31,r31,r31)     !use ref to r31 to stop simulation
        jump_reg(r0,HALT) !infinite loop

        . = 200 $

XARRAY: constant(100)
        constant(200)

        . = 400 $

YARRAY: constant(300)
        constant(400)
end
```

B.2.2. Dot Product

```
!     dot product (fortran version)
!
!     p = p + (x [i] * y[i])
!
!     note: assume word addressed machine; ie, dbl precision operands
!     reqire index to be shifted by 1 to make offset
!

include spurarch.def $

begin
  . = 0 $

START:  sub(r0,r0,r0)  !clear r0 index
        add(r10,r0,8)  !initialize loopcount
        sll(r2,r0,1)   !make offset for x array

LOOP:   ld_dbl_r(f1,r2,XARRAY) !load f1 with x data
        sll(r3,r0,1)   !make offset for y array (fpunoop)
        fpunoop        !wait for load x data to complete on fpu
        ld_dbl_r(f2,r3,YARRAY) !load f2 with y data
        add(r0,r0,1)   !increment index
        fpunoop        !wait for load y data to complete on fpu
        fmul(f3,f1,f2) !multiply x * y
        fadd(f4,f3,f4) !accumulate partial product
        sll(r2,r0,1)   !make offset for x array
        cmp_br_delayed(cc_ne,r0,r10,LOOP) !test index against loopcount

HALT:   sub(r31,r31,r31)     !use ref to r31 to stop simulation
        jump_reg(r0,HALT) !infinite loop

        . = 200 $

XARRAY: constant(100)
        constant(200)

        . = 400 $

YARRAY: constant(300)
        constant(400)
end
```

```
!     dot product (hand version)
!
!     p = p + (x [i] * y[i])
!
!     note#1: assume word addressed machine; ie, dbl precision operands
!     reqire index to be shifted by 1 to make offset
!
!     note#2: unroll loop once (ie, two computations each time through
!     loop to make cache miss start earlier
!

include spurarch.def $
begin
          . = 0 $

START:  sub(r0,r0,r0)          !clear r0 index
          add(r10,r0,4)        !initialize loopcount
          add(r2,r0,XARRAY)        !make x array base address
          add(r3,r0,YARRAY)        !make y array base address
          ld_dbl_r(f1,r2,0)    !get first x operand
          ld_dbl_r(f2,r3,0)    !get first y operand

LOOP:   fmul(f3,f1,f2)         !multiply x0*y0
          ld_dbl_r(f4,r2,2)    !load fx1 with x data
          ld_dbl_r(f5,r3,2)    !load fy1 with y data
          add(r2,r2,4)         !make new x array address
          add(r3,r3,4)         !make new y array address
          ld_dbl_r(f14,r2,4)   !start early to get block in and bury miss time
          ld_dbl_r(f4,r2,2)    !load fx1 with x data
          ld_dbl_r(f5,r3,2)    !load fy1 with y data
          fadd(f7,f3,f7)       !form new partial product
          add(r0,r0,1)         !increment loop counter
          fmul(f6,f4,f5)       !multiply x1*y1
          fadd(f7,f6,f7)       !form new partial product
          cmp_br_delayed(cc_ne,r0,r10,LOOP)  !test index against loopcount

HALT:   sub(r31,r31,r31)       !use ref to r31 to stop simulation
          jump_reg(r0,HALT) !infinite loop

          . = 200 $

XARRAY:  constant(100)
          constant(200)

          . = 400 $

YARRAY:  constant(300)
          constant(400)
end
```

B.2.3. Polynomial Evaluation

```
!     polynomial evaluation (fortran version)
!
!     p = p*k + c[i]
!     f(x) = a0*x^0 + a1*x^1 +a2*x^2 +a3*x^3
!     f(x) = a0 + [a1 + {a2 + (a3*x)}*x]*x
!
```

```
include spurarch.def $
begin
        . = 0 $

START:  sub(r0,r0,r0)          !clear r0 index
        add(r10,r0,8)          !initialize loopcount

LOOP:   sll(r3,r0,1)           !make offset for c array
        ld_dbl_r(f1,r3,CARRAY) !load f1 with c data
        add(r0,r0,1)           !increment index
        fpunoop                !wait for ld of c data to finish
        fadd(f2,f1,f3)         !c + previous partial  ==> temp (in f2)
        fmul(f3,f4,f2)         !x * temp ==> partial
        cmp_br_delayed(cc_ne,r0,r10,LOOP)  !test loop variable

HALT:   sub(r31,r31,r31)       !use ref to r31 to stop simulation
        jump_reg(r0,HALT) !infinite loop

        . = 200 $

CARRAY: constant(300)
        constant(400)
end
```

```
!     polynomial evaluation (hand version)
!
!     p = p*k + c[i]
!     f(x) = a0*x^0 + a1*x^1 +a2*x^2 +a3*x^3
!     f(x) = a0 + [a1 + {a2 + (a3*x)}*x]*x
!

include spurarch.def $
begin
        . = 0 $

START:  sub(r0,r0,r0)          !clear r0 index
        add(r10,r0,16)         !initialize loopcount
        sll(r3,r0,1)           !make offset for c array

LOOP:   fmul(f3,f4,f2)         !x * temp ==> partial
        ld_dbl_r(f1,r3,CARRAY) !load f1 with c data
        add(r3,r3,2)           !increment index
        fadd(f2,f1,f3)         !c + previous partial  ==> temp (in f2)
        cmp_br_delayed(cc_ne,r3,r10,LOOP)  !test loop variable

HALT:   sub(r31,r31,r31)       !use ref to r31 to stop simulation
        jump_reg(r0,HALT) !infinite loop

        . = 200 $

CARRAY: constant(300)
        constant(400)
end
```

B.3. Floating-point Performance for Intel, Motorola, and SPUR on Microbenchmarks

The tables below list the performance results from running the microbenchmarks on the models of the Intel i80286/i80287, Motorola MC68020/MC68881, and SPUR systems. Each of the tables indicate (both for sequential and parallel operation) the following:

- the bus width between the FPU and memory system,
- the cache-miss service time in nanoseconds,
- the total number of execution cycles expended by the FPU running this program,
- the total number of cycles for the entire program (typically 4 iterations of the loop — enough to cause one cache miss and amortize it over the rest of the loop iteration),
- the ratio of total cycles for this version of the program program to the version of the program that consumed the most cycles (both sequential and parallel),
- the percentage of total execution time that is accounted for as cache overhead (as defined in Chapter 4),
- the percentage of total execution time that is accounted for as loop overhead,
- the percentage of total execution time that is accounted for as floating-point overhead, and
- the percentage of total execution time that is accounted for as floating-point operation.

B.3.1. Intel Floating-point Performance

Table B-1. Simulation Results for Intel Running GE COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	2296	6080 (0.97)	0.0	3.8	58.4	37.8	6080 (0.97)	0.0	3.8	58.4	37.8
8	300	2296	6086 (0.97)	0.1	3.8	58.4	37.7	6086 (0.97)	0.1	3.8	58.4	37.7
8	600	2296	6092 (0.97)	0.2	3.8	58.3	37.7	6092 (0.97)	0.2	3.8	58.3	37.7
8	1200	2296	6104 (0.97)	0.4	3.8	58.2	37.6	6104 (0.97)	0.4	3.8	58.2	37.6
8	2400	2296	6128 (0.98)	0.8	3.8	58.0	37.5	6128 (0.98)	0.8	3.8	58.0	37.5
8	4800	2296	6176 (0.98)	1.6	3.8	57.5	37.2	6176 (0.98)	1.6	3.8	57.5	37.2
8	9600	2296	6272 (1.00)	3.1	3.7	56.6	36.6	6272 (1.00)	3.1	3.7	56.6	36.6
16	0	2296	4800 (0.77)	0.0	4.8	47.3	47.8	4800 (0.77)	0.0	4.8	47.3	47.8
16	300	2296	4806 (0.77)	0.1	4.8	47.3	47.8	4806 (0.77)	0.1	4.8	47.3	47.8
16	600	2296	4812 (0.77)	0.2	4.8	47.2	47.7	4812 (0.77)	0.2	4.8	47.2	47.7
16	1200	2296	4824 (0.77)	0.5	4.8	47.1	47.6	4824 (0.77)	0.5	4.8	47.1	47.6
16	2400	2296	4848 (0.77)	1.0	4.8	46.9	47.4	4848 (0.77)	1.0	4.8	46.9	47.4
16	4800	2296	4896 (0.78)	2.0	4.7	46.4	46.9	4896 (0.78)	2.0	4.7	46.4	46.9
16	9600	2296	4992 (0.80)	3.9	4.7	45.5	46.0	4992 (0.80)	3.9	4.7	45.5	46.0
32	0	2296	4160 (0.66)	0.0	5.6	39.2	55.2	4160 (0.66)	0.0	5.6	39.2	55.2
32	300	2296	4166 (0.66)	0.1	5.6	39.2	55.1	4166 (0.66)	0.1	5.6	39.2	55.1
32	600	2296	4172 (0.67)	0.3	5.6	39.1	55.0	4172 (0.67)	0.3	5.6	39.1	55.0
32	1200	2296	4184 (0.67)	0.6	5.5	39.0	54.9	4184 (0.67)	0.6	5.5	39.0	54.9
32	2400	2296	4208 (0.67)	1.1	5.5	38.8	54.6	4208 (0.67)	1.1	5.5	38.8	54.6
32	4800	2296	4256 (0.68)	2.3	5.5	38.3	53.9	4256 (0.68)	2.3	5.5	38.3	53.9
32	9600	2296	4352 (0.69)	4.4	5.3	37.5	52.8	4352 (0.69)	4.4	5.3	37.5	52.8
64	0	2296	3968 (0.63)	0.0	5.9	36.3	57.9	3968 (0.63)	0.0	5.9	36.3	57.9
64	300	2296	3974 (0.63)	0.2	5.8	36.2	57.8	3974 (0.63)	0.2	5.8	36.2	57.8
64	600	2296	3980 (0.63)	0.3	5.8	36.2	57.7	3980 (0.63)	0.3	5.8	36.2	57.7
64	1200	2296	3992 (0.64)	0.6	5.8	36.1	57.5	3992 (0.64)	0.6	5.8	36.1	57.5
64	2400	2296	4016 (0.64)	1.2	5.8	35.9	57.2	4016 (0.64)	1.2	5.8	35.9	57.2
64	4800	2296	4064 (0.65)	2.4	5.7	35.4	56.5	4064 (0.65)	2.4	5.7	35.4	56.5
64	9600	2296	4160 (0.66)	4.6	5.6	34.6	55.2	4160 (0.66)	4.6	5.6	34.6	55.2

Table B-2. Simulation Results for Intel Running GE ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	2296	5688 (0.97)	0.0	4.1	55.6	40.4	5528 (0.94)	0.0	1.3	57.2	41.5
8	300	2296	5694 (0.97)	0.1	4.1	55.5	40.3	5534 (0.94)	0.1	1.3	57.1	41.5
8	600	2296	5700 (0.97)	0.2	4.1	55.4	40.3	5540 (0.94)	0.2	1.3	57.0	41.4
8	1200	2296	5712 (0.97)	0.4	4.1	55.3	40.2	5552 (0.94)	0.4	1.3	56.9	41.4
8	2400	2296	5736 (0.98)	0.8	4.0	55.1	40.0	5576 (0.95)	0.9	1.3	56.7	41.2
8	4800	2296	5784 (0.98)	1.7	4.0	54.6	39.7	5624 (0.96)	1.7	1.3	56.2	40.8
8	9600	2296	5880 (1.00)	3.3	4.0	53.7	39.0	5720 (0.97)	3.4	1.3	55.2	40.1
16	0	2296	4408 (0.75)	0.0	5.3	42.6	52.1	4248 (0.72)	0.0	1.7	44.3	54.0
16	300	2296	4414 (0.75)	0.1	5.3	42.6	52.0	4254 (0.72)	0.1	1.7	44.2	54.0
16	600	2296	4420 (0.75)	0.3	5.3	42.5	51.9	4260 (0.72)	0.3	1.7	44.1	53.9
16	1200	2296	4432 (0.75)	0.5	5.2	42.4	51.8	4272 (0.73)	0.6	1.7	44.0	53.7
16	2400	2296	4456 (0.76)	1.1	5.2	42.2	51.5	4296 (0.73)	1.1	1.7	43.8	53.4
16	4800	2296	4504 (0.77)	2.1	5.2	41.7	51.0	4344 (0.74)	2.2	1.7	43.3	52.9
16	9600	2296	4600 (0.78)	4.2	5.0	40.9	49.9	4440 (0.76)	4.3	1.6	42.3	51.7
32	0	2296	3768 (0.64)	0.0	6.2	32.9	60.9	3608 (0.61)	0.0	2.0	34.4	63.6
32	300	2296	3774 (0.64)	0.2	6.2	32.9	60.8	3614 (0.61)	0.2	2.0	34.3	63.5
32	600	2296	3780 (0.64)	0.3	6.1	32.8	60.7	3620 (0.62)	0.3	2.0	34.3	63.4
32	1200	2296	3792 (0.64)	0.6	6.1	32.7	60.5	3632 (0.62)	0.7	2.0	34.1	63.2
32	2400	2296	3816 (0.65)	1.3	6.1	32.5	60.2	3656 (0.62)	1.3	2.0	33.9	62.8
32	4800	2296	3864 (0.66)	2.5	6.0	32.1	59.4	3704 (0.63)	2.6	1.9	33.5	62.0
32	9600	2296	3960 (0.67)	4.9	5.9	31.3	58.0	3800 (0.65)	5.1	1.9	32.6	60.4
64	0	2296	3576 (0.61)	0.0	6.5	29.3	64.2	3416 (0.58)	0.0	2.1	30.7	67.2
64	300	2296	3582 (0.61)	0.2	6.5	29.3	64.1	3422 (0.58)	0.2	2.1	30.6	67.1
64	600	2296	3588 (0.61)	0.3	6.5	29.2	64.0	3428 (0.58)	0.4	2.1	30.6	67.0
64	1200	2296	3600 (0.61)	0.7	6.4	29.1	63.8	3440 (0.59)	0.7	2.1	30.5	66.7
64	2400	2296	3624 (0.62)	1.3	6.4	28.9	63.4	3464 (0.59)	1.4	2.1	30.3	66.3
64	4800	2296	3672 (0.62)	2.6	6.3	28.5	62.5	3512 (0.60)	2.7	2.1	29.8	65.4
64	9600	2296	3768 (0.64)	5.1	6.2	27.8	60.9	3608 (0.61)	5.3	2.0	29.0	63.6

Table B-3. Simulation Results for Intel Running DP COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	2388	6248 (0.97)	0.0	4.7	57.0	38.2	6248 (0.97)	0.0	4.7	57.0	38.2
8	300	2388	6254 (0.97)	0.1	4.7	57.0	38.2	6254 (0.97)	0.1	4.7	57.0	38.2
8	600	2388	6260 (0.97)	0.2	4.7	56.9	38.1	6260 (0.97)	0.2	4.7	56.9	38.1
8	1200	2388	6272 (0.97)	0.4	4.7	56.8	38.1	6272 (0.97)	0.4	4.7	56.8	38.1
8	2400	2388	6296 (0.98)	0.8	4.7	56.6	37.9	6296 (0.98)	0.8	4.7	56.6	37.9
8	4800	2388	6344 (0.99)	1.5	4.7	56.2	37.6	6344 (0.99)	1.5	4.7	56.2	37.6
8	9600	2388	6440 (1.00)	3.0	4.6	55.3	37.1	6440 (1.00)	3.0	4.6	55.3	37.1
16	0	2388	4968 (0.77)	0.0	6.0	46.0	48.1	4968 (0.77)	0.0	6.0	46.0	48.1
16	300	2388	4974 (0.77)	0.1	6.0	45.9	48.0	4974 (0.77)	0.1	6.0	45.9	48.0
16	600	2388	4980 (0.77)	0.2	5.9	45.9	48.0	4980 (0.77)	0.2	5.9	45.9	48.0
16	1200	2388	4992 (0.78)	0.5	5.9	45.8	47.8	4992 (0.78)	0.5	5.9	45.8	47.8
16	2400	2388	5016 (0.78)	1.0	5.9	45.5	47.6	5016 (0.78)	1.0	5.9	45.5	47.6
16	4800	2388	5064 (0.79)	1.9	5.9	45.1	47.2	5064 (0.79)	1.9	5.9	45.1	47.2
16	9600	2388	5160 (0.80)	3.7	5.7	44.3	46.3	5160 (0.80)	3.7	5.7	44.3	46.3
32	0	2388	4328 (0.67)	0.0	6.8	38.0	55.2	4328 (0.67)	0.0	6.8	38.0	55.2
32	300	2388	4334 (0.67)	0.1	6.8	37.9	55.1	4334 (0.67)	0.1	6.8	37.9	55.1
32	600	2388	4340 (0.67)	0.3	6.8	37.9	55.0	4340 (0.67)	0.3	6.8	37.9	55.0
32	1200	2388	4352 (0.68)	0.6	6.8	37.8	54.9	4352 (0.68)	0.6	6.8	37.8	54.9
32	2400	2388	4376 (0.68)	1.1	6.8	37.6	54.6	4376 (0.68)	1.1	6.8	37.6	54.6
32	4800	2388	4424 (0.69)	2.2	6.7	37.2	54.0	4424 (0.69)	2.2	6.7	37.2	54.0
32	9600	2388	4520 (0.70)	4.3	6.6	36.4	52.8	4520 (0.70)	4.3	6.6	36.4	52.8
64	0	2388	4136 (0.64)	0.0	7.2	35.1	57.7	4136 (0.64)	0.0	7.2	35.1	57.7
64	300	2388	4142 (0.64)	0.1	7.2	35.1	57.7	4142 (0.64)	0.1	7.2	35.1	57.7
64	600	2388	4148 (0.64)	0.3	7.1	35.0	57.6	4148 (0.64)	0.3	7.1	35.0	57.6
64	1200	2388	4160 (0.65)	0.6	7.1	34.9	57.4	4160 (0.65)	0.6	7.1	34.9	57.4
64	2400	2388	4184 (0.65)	1.2	7.1	34.7	57.1	4184 (0.65)	1.2	7.1	34.7	57.1
64	4800	2388	4232 (0.66)	2.3	7.0	34.3	56.4	4232 (0.66)	2.3	7.0	34.3	56.4
64	9600	2388	4328 (0.67)	4.4	6.8	33.5	55.2	4328 (0.67)	4.4	6.8	33.5	55.2

Table B-4. Simulation Results for Intel Running DP ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	1900	4548 (0.96)	0.0	6.5	51.7	41.8	4356 (0.92)	0.0	2.4	54.0	43.6
8	300	1900	4554 (0.96)	0.1	6.5	51.6	41.7	4362 (0.92)	0.1	2.4	53.9	43.6
8	600	1900	4560 (0.96)	0.3	6.5	51.6	41.7	4368 (0.92)	0.3	2.4	53.8	43.5
8	1200	1900	4572 (0.96)	0.5	6.5	51.4	41.6	4380 (0.92)	0.5	2.4	53.7	43.4
8	2400	1900	4596 (0.97)	1.0	6.4	51.2	41.3	4404 (0.93)	1.1	2.4	53.4	43.1
8	4800	1900	4644 (0.98)	2.1	6.4	50.6	40.9	4452 (0.94)	2.2	2.3	52.8	42.7
8	9600	1900	4740 (1.00)	4.1	6.2	49.6	40.1	4548 (0.96)	4.2	2.3	51.7	41.8
16	0	1900	3588 (0.76)	0.0	8.3	38.8	53.0	3396 (0.72)	0.0	3.1	41.0	55.9
16	300	1900	3594 (0.76)	0.2	8.2	38.7	52.9	3402 (0.72)	0.2	3.1	40.9	55.8
16	600	1900	3600 (0.76)	0.3	8.2	38.7	52.8	3408 (0.72)	0.4	3.1	40.8	55.8
16	1200	1900	3612 (0.76)	0.7	8.2	38.5	52.6	3420 (0.72)	0.7	3.0	40.7	55.6
16	2400	1900	3636 (0.77)	1.3	8.1	38.3	52.3	3444 (0.73)	1.4	3.0	40.4	55.2
16	4800	1900	3684 (0.78)	2.6	8.0	37.8	51.6	3492 (0.74)	2.8	3.0	39.9	54.4
16	9600	1900	3780 (0.80)	5.1	7.8	36.8	50.3	3588 (0.76)	5.4	2.9	38.8	53.0
32	0	1900	3108 (0.66)	0.0	9.5	29.3	61.1	2916 (0.62)	0.0	3.6	31.3	65.2
32	300	1900	3114 (0.66)	0.2	9.5	29.3	61.0	2922 (0.62)	0.2	3.6	31.2	65.0
32	600	1900	3120 (0.66)	0.4	9.5	29.2	60.9	2928 (0.62)	0.4	3.6	31.1	64.9
32	1200	1900	3132 (0.66)	0.8	9.5	29.1	60.7	2940 (0.62)	0.8	3.5	31.0	64.6
32	2400	1900	3156 (0.67)	1.5	9.4	28.9	60.2	2964 (0.63)	1.6	3.5	30.8	64.1
32	4800	1900	3204 (0.68)	3.0	9.2	28.5	59.3	3012 (0.64)	3.2	3.5	30.3	63.1
32	9600	1900	3300 (0.70)	5.8	9.0	27.6	57.6	3108 (0.66)	6.2	3.4	29.3	61.1
64	0	1900	3028 (0.64)	0.0	9.8	27.5	62.7	2836 (0.60)	0.0	3.7	29.3	67.0
64	300	1900	3034 (0.64)	0.2	9.8	27.4	62.6	2842 (0.60)	0.2	3.7	29.3	66.9
64	600	1900	3040 (0.64)	0.4	9.7	27.4	62.5	2848 (0.60)	0.4	3.7	29.2	66.7
64	1200	1900	3052 (0.64)	0.8	9.7	27.3	62.3	2860 (0.60)	0.8	3.6	29.1	66.4
64	2400	1900	3076 (0.65)	1.6	9.6	27.0	61.8	2884 (0.61)	1.7	3.6	28.8	65.9
64	4800	1900	3124 (0.66)	3.1	9.5	26.6	60.8	2932 (0.62)	3.3	3.6	28.4	64.8
64	9600	1900	3220 (0.68)	6.0	9.2	25.8	59.0	3028 (0.64)	6.3	3.4	27.5	62.7

Table B-5. Simulation Results for Intel Running PE COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	2256	5936 (0.98)	0.0	1.9	60.1	38.0	5936 (0.98)	0.0	1.9	60.1	38.0
8	300	2256	5939 (0.98)	0.1	1.9	60.1	38.0	5939 (0.98)	0.1	1.9	60.1	38.0
8	600	2256	5942 (0.99)	0.1	1.9	60.0	38.0	5942 (0.99)	0.1	1.9	60.0	38.0
8	1200	2256	5948 (0.99)	0.2	1.9	60.0	37.9	5948 (0.99)	0.2	1.9	60.0	37.9
8	2400	2256	5960 (0.99)	0.4	1.9	59.9	37.9	5960 (0.99)	0.4	1.9	59.9	37.9
8	4800	2256	5984 (0.99)	0.8	1.9	59.6	37.7	5984 (0.99)	0.8	1.9	59.6	37.7
8	9600	2256	6032 (1.00)	1.6	1.9	59.2	37.4	6032 (1.00)	1.6	1.9	59.2	37.4
16	0	2256	4656 (0.77)	0.0	2.4	49.1	48.5	4656 (0.77)	0.0	**2.4**	**49.1**	48.5
16	300	2256	4659 (0.77)	0.1	2.4	49.1	48.4	4659 (0.77)	0.1	**2.4**	**49.1**	48.4
16	600	2256	4662 (0.77)	0.1	2.4	49.1	48.4	4662 (0.77)	0.1	2.4	49.1	48.4
16	1200	2256	4668 (0.77)	0.3	2.4	49.0	48.3	4668 (0.77)	0.3	2.4	49.0	48.3
16	2400	2256	4680 (0.78)	0.5	2.4	48.9	48.2	4680 (0.78)	0.5	2.4	48.9	48.2
16	4800	2256	4704 (0.78)	1.0	2.4	48.6	48.0	4704 (0.78)	1.0	2.4	48.6	48.0
16	9600	2256	4752 (0.79)	2.0	2.4	48.1	47.5	4752 (0.79)	2.0	2.4	48.1	47.5
32	0	2256	4016 (0.67)	0.0	2.8	41.0	56.2	4016 (0.67)	0.0	2.8	41.0	56.2
32	300	2256	4019 (0.67)	0.1	2.8	41.0	56.1	4019 (0.67)	0.1	2.8	41.0	**56.1**
32	600	2256	4022 (0.67)	0.1	2.8	41.0	56.1	4022 (0.67)	0.1	2.8	41.0	**56.1**
32	1200	2256	4028 (0.67)	0.3	2.8	40.9	56.0	4028 (0.67)	0.3	2.8	40.9	56.0
32	2400	2256	4040 (0.67)	0.6	2.8	40.8	55.8	4040 (0.67)	0.6	2.8	40.8	55.8
32	4800	2256	4064 (0.67)	1.2	2.8	40.6	55.5	4064 (0.67)	1.2	2.8	40.6	55.5
32	9600	2256	4112 (0.68)	2.3	2.7	40.1	54.9	4112 (0.68)	2.3	2.7	40.1	54.9
64	0	2256	3824 (0.63)	0.0	2.9	38.1	59.0	3824 (0.63)	0.0	2.9	38.1	59.0
64	300	2256	3827 (0.63)	0.1	2.9	38.0	58.9	3827 (0.63)	0.1	2.9	38.0	58.9
64	600	2256	3830 (0.63)	0.2	2.9	38.0	58.9	3830 (0.63)	0.2	2.9	38.0	58.9
64	1200	2256	3836 (0.64)	0.3	2.9	38.0	58.8	3836 (0.64)	0.3	2.9	38.0	58.8
64	2400	2256	3848 (0.64)	0.6	2.9	37.8	58.6	3848 (0.64)	0.6	2.9	37.8	58.6
64	4800	2256	3872 (0.64)	1.2	2.9	37.6	58.3	3872 (0.64)	1.2	2.9	37.6	58.3
64	9600	2256	3920 (0.65)	2.5	2.9	37.1	57.6	3920 (0.65)	2.5	2.9	37.1	57.6

Table B-6. Simulation Results for Intel Running PE ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	1748	4228 (0.98)	0.0	2.7	56.0	41.3	4188 (0.97)	0.0	1.7	56.5	41.7
8	300	1748	4231 (0.98)	0.1	2.7	56.0	41.3	4191 (0.97)	0.1	1.7	56.5	41.7
8	600	1748	4234 (0.98)	0.1	2.7	55.9	41.3	4194 (0.97)	0.1	1.7	56.5	41.7
8	1200	1748	4240 (0.98)	0.3	2.6	55.8	41.2	4200 (0.97)	0.3	1.7	56.4	41.6
8	2400	1748	4252 (0.98)	0.6	2.6	55.7	41.1	4212 (0.97)	0.6	1.7	56.2	41.5
8	4800	1748	4276 (0.99)	1.1	2.6	55.4	40.9	4236 (0.98)	1.1	1.7	55.9	41.3
8	9600	1748	4324 (1.00)	2.2	2.6	54.8	40.4	4284 (0.99)	2.2	1.7	55.3	40.8
16	0	1748	3268 (0.76)	0.0	3.4	43.1	53.5	3228 (0.75)	0.0	2.2	43.6	54.2
16	300	1748	3271 (0.76)	0.1	3.4	43.0	53.4	3231 (0.75)	0.1	2.2	43.6	54.1
16	600	1748	3274 (0.76)	0.2	3.4	43.0	53.4	3234 (0.75)	0.2	2.2	43.5	54.1
16	1200	1748	3280 (0.76)	0.4	3.4	42.9	53.3	3240 (0.75)	0.4	2.2	43.5	54.0
16	2400	1748	3292 (0.76)	0.7	3.4	42.8	53.1	3252 (0.75)	0.7	2.2	43.3	53.8
16	4800	1748	3316 (0.77)	1.5	3.4	42.5	52.7	3276 (0.76)	1.5	2.2	43.0	53.4
16	9600	1748	3364 (0.78)	2.9	3.3	41.9	52.0	3324 (0.77)	2.9	2.2	42.4	52.6
32	0	1748	2788 (0.64)	0.0	4.0	33.3	62.7	2748 (0.64)	0.0	2.6	33.8	63.6
32	300	1748	2791 (0.65)	0.1	4.0	33.2	62.6	2751 (0.64)	0.1	2.6	33.7	63.5
32	600	1748	2794 (0.65)	0.2	4.0	33.2	62.6	2754 (0.64)	0.2	2.6	33.7	63.5
32	1200	1748	2800 (0.65)	0.4	4.0	33.1	62.4	2760 (0.64)	0.4	2.6	33.6	63.3
32	2400	1748	2812 (0.65)	0.9	4.0	33.0	62.2	2772 (0.64)	0.9	2.6	33.5	63.1
32	4800	1748	2836 (0.66)	1.7	4.0	32.7	61.6	2796 (0.65)	1.7	2.6	33.2	62.5
32	9600	1748	2884 (0.67)	3.3	3.9	32.2	60.6	2844 (0.66)	3.4	2.5	32.6	61.5
64	0	1748	2708 (0.63)	0.0	4.1	31.3	64.5	2668 (0.62)	0.0	2.7	31.8	65.5
64	300	1748	2711 (0.63)	0.1	4.1	31.3	64.5	2671 (0.62)	0.1	2.7	31.7	65.4
64	600	1748	2714 (0.63)	0.2	4.1	31.2	64.4	2674 (0.62)	0.2	2.7	31.7	65.4
64	1200	1748	2720 (0.63)	0.4	4.1	31.2	64.3	2680 (0.62)	0.4	2.7	31.6	65.2
64	2400	1748	2732 (0.63)	0.9	4.1	31.0	64.0	2692 (0.62)	0.9	2.7	31.5	64.9
64	4800	1748	2756 (0.64)	1.7	4.1	30.8	63.4	2716 (0.63)	1.8	2.7	31.2	64.4
64	9600	1748	2804 (0.65)	3.4	4.0	30.2	62.3	2764 (0.64)	3.5	2.6	30.7	63.2

B.3.2. Motorola Floating-point Performance

Table B-7. Simulation Results for Motorola Running GE COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	820	3916 (0.92)	0.0	9.2	69.9	20.9	3096 (0.73)	0.0	11.6	61.9	26.5
8	300	820	3926 (0.93)	0.3	9.2	69.6	20.9	3106 (0.73)	0.3	11.6	61.7	26.4
8	600	820	3936 (0.93)	0.5	9.2	69.5	20.8	3116 (0.74)	0.6	11.6	61.5	26.3
8	1200	820	3956 (0.93)	1.0	9.1	69.1	20.7	3136 (0.74)	1.3	11.5	61.1	26.1
8	2400	820	3996 (0.94)	2.0	9.0	68.5	20.5	3176 (0.75)	2.5	11.3	60.4	25.8
8	4800	820	4076 (0.96)	3.9	8.8	67.2	20.1	3256 (0.77)	4.9	11.1	58.8	25.2
8	9600	820	4236 (1.00)	7.6	8.5	64.6	19.4	3416 (0.81)	9.4	10.5	56.1	24.0
16	0	820	2776 (0.66)	0.0	8.4	62.1	29.5	1956 (0.46)	0.0	11.9	46.2	41.9
16	300	820	2786 (0.66)	0.4	8.3	61.9	29.4	1966 (0.46)	0.5	11.8	46.0	41.7
16	600	820	2796 (0.66)	0.7	8.3	61.7	29.3	1976 (0.47)	1.0	11.7	45.8	41.5
16	1200	820	2816 (0.66)	1.4	8.2	61.2	29.1	1996 (0.47)	2.0	11.6	45.3	41.1
16	2400	820	2856 (0.67)	2.8	8.1	60.4	28.7	2036 (0.48)	3.9	11.4	44.4	40.3
16	4800	820	2936 (0.69)	5.5	7.9	58.7	27.9	2116 (0.50)	7.6	11.0	42.7	38.8
16	9600	820	3096 (0.73)	10.3	7.5	55.7	26.5	2276 (0.54)	14.1	10.2	39.8	36.0
32	0	820	2428 (0.57)	0.0	6.9	59.4	33.8	1608 (0.38)	0.0	10.4	38.5	51.0
32	300	820	2438 (0.58)	0.4	6.9	59.1	33.6	1618 (0.38)	0.6	10.4	38.3	50.7
32	600	820	2448 (0.58)	0.8	6.9	58.8	33.5	1628 (0.38)	1.2	10.3	38.0	50.4
32	1200	820	2468 (0.58)	1.6	6.8	58.3	33.2	1648 (0.39)	2.4	10.2	37.7	49.8
32	2400	820	2508 (0.59)	3.2	6.7	57.4	32.7	1688 (0.40)	4.7	10.0	36.7	48.6
32	4800	820	2588 (0.61)	6.2	6.5	55.6	31.7	1768 (0.42)	9.1	9.5	35.1	46.4
32	9600	820	2748 (0.65)	11.6	6.1	52.4	29.8	1928 (0.46)	16.6	8.7	32.2	42.5
64	0	820	2296 (0.54)	0.0	7.3	57.0	35.7	1476 (0.35)	0.0	11.4	33.0	55.6
64	300	820	2306 (0.54)	0.4	7.3	56.7	35.6	1486 (0.35)	0.7	11.3	32.9	55.2
64	600	820	2316 (0.55)	0.9	7.3	56.5	35.4	1496 (0.35)	1.3	11.2	32.6	54.8
64	1200	820	2336 (0.55)	1.7	7.2	56.0	35.1	1516 (0.36)	2.6	11.1	32.2	54.1
64	2400	820	2376 (0.56)	3.4	7.1	55.0	34.5	1556 (0.37)	5.1	10.8	31.3	52.7
64	4800	820	2456 (0.58)	6.5	6.8	53.2	33.4	1636 (0.39)	9.8	10.3	29.8	50.1
64	9600	820	2616 (0.62)	12.2	6.4	50.0	31.3	1796 (0.42)	17.8	9.4	27.2	45.7

Table B-8. Simulation Results for Motorola Running GE ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	820	3572 (0.92)	0.0	7.3	69.7	23.0	2752 (0.71)	0.0	9.5	60.8	29.8
8	300	820	3582 (0.92)	0.3	7.3	69.6	22.9	2762 (0.71)	0.4	9.4	60.5	29.7
8	600	820	3592 (0.92)	0.6	7.2	69.4	22.8	2772 (0.71)	0.7	9.4	60.3	29.6
8	1200	820	3612 (0.93)	1.1	7.2	69.0	22.7	2792 (0.72)	1.4	9.3	59.9	29.4
8	2400	820	3652 (0.94)	2.2	7.1	68.2	22.5	2832 (0.73)	2.8	9.2	59.0	29.0
8	4800	820	3732 (0.96)	4.3	7.0	66.7	22.0	2912 (0.75)	5.5	8.9	57.4	28.2
8	9600	820	3892 (1.00)	8.2	6.7	64.1	21.1	3072 (0.79)	10.4	8.5	54.4	26.7
16	0	820	2552 (0.66)	0.0	5.8	62.0	32.1	1732 (0.45)	0.0	8.6	44.1	47.3
16	300	820	2562 (0.66)	0.4	5.8	61.8	32.0	1742 (0.45)	0.6	8.5	43.9	47.1
16	600	820	2572 (0.66)	0.8	5.8	61.6	31.9	1752 (0.45)	1.1	8.5	43.6	46.8
16	1200	820	2592 (0.67)	1.5	5.7	61.1	31.6	1772 (0.46)	2.3	8.4	43.1	46.3
16	2400	820	2632 (0.68)	3.0	5.6	60.2	31.2	1812 (0.47)	4.4	8.2	42.2	45.3
16	4800	820	2712 (0.70)	5.9	5.5	58.4	30.2	1892 (0.49)	8.5	7.8	40.4	43.3
16	9600	820	2872 (0.74)	11.1	5.2	55.2	28.6	2052 (0.53)	15.6	7.2	37.3	40.0
32	0	820	2264 (0.58)	0.0	4.1	59.7	36.2	1444 (0.37)	0.0	6.4	36.9	56.8
32	300	820	2274 (0.58)	0.4	4.1	59.5	36.1	1454 (0.37)	0.7	6.3	36.6	56.4
32	600	820	2284 (0.59)	0.9	4.0	59.2	35.9	1464 (0.38)	1.4	6.3	36.4	56.0
32	1200	820	2304 (0.59)	1.7	4.0	58.7	35.6	1484 (0.38)	2.7	6.2	35.9	55.3
32	2400	820	2344 (0.60)	3.4	3.9	57.7	35.0	1524 (0.39)	5.3	6.0	34.9	53.8
32	4800	820	2424 (0.62)	6.6	3.8	55.8	33.8	1604 (0.41)	10.0	5.7	33.1	51.1
32	9600	820	2584 (0.66)	12.4	3.6	52.4	31.7	1764 (0.45)	18.1	5.2	30.1	46.5
64	0	820	2156 (0.55)	0.0	4.3	57.7	38.0	1336 (0.34)	0.0	6.9	31.8	61.4
64	300	820	2166 (0.56)	0.5	4.3	57.4	37.9	1346 (0.35)	0.7	6.8	31.5	60.9
64	600	820	2176 (0.56)	0.9	4.2	57.2	37.7	1356 (0.35)	1.5	6.8	31.3	60.5
64	1200	820	2196 (0.56)	1.8	4.2	56.7	37.3	1376 (0.35)	2.9	6.7	30.9	59.6
64	2400	820	2236 (0.57)	3.6	4.1	55.7	36.7	1416 (0.36)	5.7	6.5	29.9	57.9
64	4800	820	2316 (0.60)	6.9	4.0	53.7	35.4	1496 (0.38)	10.7	6.2	28.3	54.8
64	9600	820	2476 (0.64)	12.9	3.7	50.2	33.1	1656 (0.43)	19.3	5.6	25.6	49.5

Table B-9. Simulation Results for Motorola Running DP COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	728	3884 (0.92)	0.0	9.8	71.4	18.7	3156 (0.75)	0.0	12.0	64.9	23.1
8	300	728	3894 (0.93)	0.3	9.8	71.3	18.7	3166 (0.75)	0.3	12.0	64.6	23.0
8	600	728	3904 (0.93)	0.5	9.7	71.1	18.6	3176 (0.76)	0.6	12.0	64.5	22.9
8	1200	728	3924 (0.93)	1.0	9.7	70.7	18.6	3196 (0.76)	1.3	11.9	64.0	22.8
8	2400	728	3964 (0.94)	2.0	9.6	70.0	18.4	3236 (0.77)	2.5	11.7	63.3	22.5
8	4800	728	4044 (0.96)	4.0	9.4	68.7	18.0	3316 (0.79)	4.8	11.5	61.7	22.0
8	9600	728	4204 (1.00)	7.6	9.0	66.1	17.3	3476 (0.83)	9.2	10.9	58.9	20.9
16	0	728	2696 (0.64)	0.0	9.4	63.6	27.0	1968 (0.47)	0.0	12.8	50.2	37.0
16	300	728	2706 (0.64)	0.4	9.3	63.5	26.9	1978 (0.47)	0.5	12.7	50.0	36.8
16	600	728	2716 (0.65)	0.7	9.3	63.2	26.8	1988 (0.47)	1.0	12.7	49.7	36.6
16	1200	728	2736 (0.65)	1.5	9.2	62.7	26.6	2008 (0.48)	2.0	12.5	49.2	36.3
16	2400	728	2776 (0.66)	2.9	9.1	61.8	26.2	2048 (0.49)	3.9	12.3	48.3	35.5
16	4800	728	2856 (0.68)	5.6	8.8	60.1	25.5	2128 (0.51)	7.5	11.8	46.4	34.2
16	9600	728	3016 (0.72)	10.6	8.4	56.9	24.1	2288 (0.54)	14.0	11.0	43.2	31.8
32	0	728	2324 (0.55)	0.0	8.1	60.6	31.3	1596 (0.38)	0.0	11.8	42.6	45.6
32	300	728	2334 (0.56)	0.4	8.1	60.3	31.2	1606 (0.38)	0.6	11.7	42.3	45.3
32	600	728	2344 (0.56)	0.9	8.0	60.1	31.1	1616 (0.38)	1.2	11.6	42.1	45.0
32	1200	728	2364 (0.56)	1.7	8.0	59.6	30.8	1636 (0.39)	2.4	11.5	41.6	44.5
32	2400	728	2404 (0.57)	3.3	7.8	58.5	30.3	1676 (0.40)	4.8	11.2	40.6	43.4
32	4800	728	2484 (0.59)	6.4	7.6	56.6	29.3	1756 (0.42)	9.1	10.7	38.7	41.5
32	9600	728	2644 (0.63)	12.1	7.1	53.2	27.5	1916 (0.46)	16.7	9.8	35.5	38.0
64	0	728	2180 (0.52)	0.0	8.6	58.0	33.4	1452 (0.35)	0.0	12.9	36.9	50.1
64	300	728	2190 (0.52)	0.5	8.6	57.8	33.2	1462 (0.35)	0.7	12.9	36.6	49.8
64	600	728	2200 (0.52)	0.9	8.6	57.5	33.1	1472 (0.35)	1.4	12.8	36.4	49.5
64	1200	728	2220 (0.53)	1.8	8.5	57.0	32.8	1492 (0.35)	2.7	12.6	36.0	48.8
64	2400	728	2260 (0.54)	3.5	8.3	55.9	32.2	1532 (0.36)	5.2	12.3	35.0	47.5
64	4800	728	2340 (0.56)	6.8	8.0	54.0	31.1	1612 (0.38)	9.9	11.7	33.3	45.2
64	9600	728	2500 (0.59)	12.8	7.5	50.5	29.1	1772 (0.42)	18.1	10.6	30.3	41.1

Table B-10. Simulation Results for Motorola Running DP ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	522	2202 (0.87)	0.0	10.2	66.1	23.7	1680 (0.67)	0.0	7.6	61.4	31.1
8	300	522	2212 (0.88)	0.5	10.1	65.8	23.6	1690 (0.67)	0.6	7.6	61.0	30.9
8	600	522	2222 (0.88)	0.9	10.1	65.5	23.5	1700 (0.67)	1.2	7.5	60.6	30.7
8	1200	522	2242 (0.89)	1.8	10.0	65.0	23.3	1720 (0.68)	2.3	7.4	59.9	30.3
8	2400	522	2282 (0.90)	3.5	9.8	63.8	22.9	1760 (0.70)	4.6	7.3	58.6	29.7
8	4800	522	2362 (0.94)	6.8	9.5	61.7	22.1	1840 (0.73)	8.7	7.0	56.0	28.4
8	9600	522	2522 (1.00)	12.7	8.9	57.7	20.7	2000 (0.79)	16.0	6.4	51.5	26.1
16	0	522	1570 (0.62)	0.0	10.2	56.6	33.2	1048 (0.42)	0.0	6.1	44.1	49.8
16	300	522	1580 (0.63)	0.6	10.1	56.2	33.0	1058 (0.42)	0.9	6.1	43.6	49.3
16	600	522	1590 (0.63)	1.3	10.1	55.8	32.8	1068 (0.42)	1.9	6.0	43.3	48.9
16	1200	522	1610 (0.64)	2.5	9.9	55.2	32.4	1088 (0.43)	3.7	5.9	42.5	48.0
16	2400	522	1650 (0.65)	4.9	9.7	53.8	31.6	1128 (0.45)	7.1	5.7	40.9	46.3
16	4800	522	1730 (0.69)	9.3	9.3	51.4	30.2	1208 (0.48)	13.2	5.3	38.2	43.2
16	9600	522	1890 (0.75)	16.9	8.5	47.0	27.6	1368 (0.54)	23.4	4.7	33.8	38.2
32	0	522	1402 (0.56)	0.0	9.1	53.6	37.2	880 (0.35)	0.0	3.6	37.0	59.3
32	300	522	1412 (0.56)	0.7	9.1	53.2	37.0	890 (0.35)	1.1	3.6	36.6	58.7
32	600	522	1422 (0.56)	1.4	9.0	52.9	36.7	900 (0.36)	2.2	3.6	36.3	58.0
32	1200	522	1442 (0.57)	2.8	8.9	52.1	36.2	920 (0.36)	4.4	3.5	35.4	56.7
32	2400	522	1482 (0.59)	5.4	8.6	50.7	35.2	960 (0.38)	8.3	3.3	34.0	54.4
32	4800	522	1562 (0.62)	10.2	8.2	48.2	33.4	1040 (0.41)	15.4	3.1	31.3	50.2
32	9600	522	1722 (0.68)	18.6	7.4	43.7	30.3	1200 (0.48)	26.7	2.7	27.2	43.5
64	0	522	1338 (0.53)	0.0	9.6	51.4	39.0	816 (0.32)	0.0	3.9	32.1	64.0
64	300	522	1348 (0.53)	0.7	9.5	51.0	38.7	826 (0.33)	1.2	3.9	31.7	63.2
64	600	522	1358 (0.54)	1.5	9.4	50.7	38.4	836 (0.33)	2.4	3.8	31.3	62.4
64	1200	522	1378 (0.55)	2.9	9.3	49.9	37.9	856 (0.34)	4.7	3.7	30.6	61.0
64	2400	522	1418 (0.56)	5.6	9.0	48.6	36.8	896 (0.36)	8.9	3.6	29.3	58.3
64	4800	522	1498 (0.59)	10.7	8.5	46.0	34.8	976 (0.39)	16.4	3.3	26.9	53.5
64	9600	522	1658 (0.66)	19.3	7.7	41.5	31.5	1136 (0.45)	28.2	2.8	23.1	46.0

Table B-11. Simulation Results for Motorola Running PE COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	520	1896 (0.92)	0.0	7.6	64.9	27.4	1376 (0.67)	0.0	10.5	51.8	37.8
8	300	520	1901 (0.92)	0.3	7.6	64.8	27.4	1381 (0.67)	0.4	10.4	51.5	37.7
8	600	520	1906 (0.93)	0.5	7.6	64.7	27.3	1386 (0.67)	0.7	10.4	51.4	37.5
8	1200	520	1916 (0.93)	1.0	7.5	64.3	27.1	1396 (0.68)	1.4	10.3	51.0	37.2
8	2400	520	1936 (0.94)	2.1	7.4	63.6	26.9	1416 (0.69)	2.8	10.2	50.3	36.7
8	4800	520	1976 (0.96)	4.1	7.3	62.4	26.3	1456 (0.71)	5.5	9.9	48.9	35.7
8	9600	520	2056 (1.00)	7.8	7.0	59.9	25.3	1536 (0.75)	10.4	9.4	46.4	33.9
16	0	520	1436 (0.70)	0.0	7.8	56.0	36.2	916 (0.45)	0.0	12.2	31.0	56.8
16	300	520	1441 (0.70)	0.3	7.8	55.8	36.1	921 (0.45)	0.5	12.2	30.8	56.5
16	600	520	1446 (0.70)	0.7	7.8	55.6	36.0	926 (0.45)	1.1	12.1	30.7	56.2
16	1200	520	1456 (0.71)	1.4	7.7	55.2	35.7	936 (0.46)	2.1	12.0	30.3	55.6
16	2400	520	1476 (0.72)	2.7	7.6	54.5	35.2	956 (0.46)	4.2	11.7	29.7	54.4
16	4800	520	1516 (0.74)	5.3	7.4	53.1	34.3	996 (0.48)	8.0	11.2	28.5	52.2
16	9600	520	1596 (0.78)	10.0	7.0	50.4	32.6	1076 (0.52)	14.9	10.4	26.4	48.3
32	0	520	1352 (0.66)	0.0	7.1	54.4	38.5	832 (0.40)	0.0	11.5	26.0	62.5
32	300	520	1357 (0.66)	0.4	7.1	54.2	38.3	837 (0.41)	0.6	11.5	25.8	62.1
32	600	520	1362 (0.66)	0.7	7.1	54.1	38.2	842 (0.41)	1.2	11.4	25.7	61.8
32	1200	520	1372 (0.67)	1.5	7.0	53.6	37.9	852 (0.41)	2.4	11.3	25.4	61.0
32	2400	520	1392 (0.68)	2.9	6.9	52.9	37.4	872 (0.42)	4.6	11.0	24.8	59.6
32	4800	520	1432 (0.70)	5.6	6.7	51.4	36.3	912 (0.44)	8.8	10.5	23.7	57.0
32	9600	520	1512 (0.74)	10.6	6.4	48.7	34.4	992 (0.48)	16.1	9.7	21.8	52.4
64	0	520	1320 (0.64)	0.0	7.3	53.3	39.4	800 (0.39)	0.0	12.0	23.0	65.0
64	300	520	1325 (0.64)	0.4	7.3	53.1	39.2	805 (0.39)	0.6	11.9	22.8	64.6
64	600	520	1330 (0.65)	0.8	7.2	53.0	39.1	810 (0.39)	1.2	11.9	22.7	64.2
64	1200	520	1340 (0.65)	1.5	7.2	52.5	38.8	820 (0.40)	2.4	11.7	22.5	63.4
64	2400	520	1360 (0.66)	2.9	7.1	51.8	38.2	840 (0.41)	4.8	11.4	21.9	61.9
64	4800	520	1400 (0.68)	5.7	6.9	50.2	37.1	880 (0.43)	9.1	10.9	20.9	59.1
64	9600	520	1480 (0.72)	10.8	6.5	47.6	35.1	960 (0.47)	16.7	10.0	19.2	54.2

Table B-12. Simulation Results for Motorola Running PE ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	520	1844 (0.92)	0.0	5.6	66.1	28.2	1324 (0.66)	0.0	7.9	52.9	39.3
8	300	520	1849 (0.92)	0.3	5.6	66.0	28.1	1329 (0.66)	0.4	7.8	52.7	39.1
8	600	520	1854 (0.93)	0.5	5.6	65.8	28.0	1334 (0.67)	0.8	7.8	52.5	39.0
8	1200	520	1864 (0.93)	1.1	5.6	65.5	27.9	1344 (0.67)	1.5	7.7	52.1	38.7
8	2400	520	1884 (0.94)	2.1	5.5	64.7	27.6	1364 (0.68)	2.9	7.6	51.3	38.1
8	4800	520	1924 (0.96)	4.2	5.4	63.4	27.0	1404 (0.70)	5.7	7.4	49.9	37.0
8	9600	520	2004 (1.00)	8.0	5.2	60.9	25.9	1484 (0.74)	10.8	7.0	47.2	35.0
16	0	520	1384 (0.69)	0.0	5.2	57.3	37.6	864 (0.43)	0.0	8.3	31.4	60.2
16	300	520	1389 (0.69)	0.4	5.2	57.0	37.4	869 (0.43)	0.6	8.3	31.3	59.8
16	600	520	1394 (0.70)	0.7	5.2	56.8	37.3	874 (0.44)	1.1	8.2	31.1	59.5
16	1200	520	1404 (0.70)	1.4	5.1	56.4	37.0	884 (0.44)	2.3	8.1	30.7	58.8
16	2400	520	1424 (0.71)	2.8	5.1	55.6	36.5	904 (0.45)	4.4	8.0	30.1	57.5
16	4800	520	1464 (0.73)	5.5	4.9	54.1	35.5	944 (0.47)	8.5	7.6	28.8	55.1
16	9600	520	1544 (0.77)	10.4	4.7	51.3	33.7	1024 (0.51)	15.6	7.0	26.5	50.8
32	0	520	1300 (0.65)	0.0	4.3	55.7	40.0	780 (0.39)	0.0	7.2	26.1	66.7
32	300	520	1305 (0.65)	0.4	4.3	55.4	39.8	785 (0.39)	0.6	7.1	26.0	66.2
32	600	520	1310 (0.65)	0.8	4.3	55.3	39.7	790 (0.39)	1.3	7.1	25.8	65.8
32	1200	520	1320 (0.66)	1.5	4.2	54.8	39.4	800 (0.40)	2.5	7.0	25.5	65.0
32	2400	520	1340 (0.67)	3.0	4.2	54.0	38.8	820 (0.41)	4.9	6.8	24.9	63.4
32	4800	520	1380 (0.69)	5.8	4.1	52.4	37.7	860 (0.43)	9.3	6.5	23.7	60.5
32	9600	520	1460 (0.73)	11.0	3.8	49.6	35.6	940 (0.47)	17.0	6.0	21.7	55.3
64	0	520	1268 (0.63)	0.0	4.4	54.5	41.0	748 (0.37)	0.0	7.5	23.0	69.5
64	300	520	1273 (0.64)	0.4	4.4	54.4	40.8	753 (0.38)	0.7	7.4	22.9	69.1
64	600	520	1278 (0.64)	0.8	4.4	54.2	40.7	758 (0.38)	1.3	7.4	22.7	68.6
64	1200	520	1288 (0.64)	1.6	4.4	53.7	40.4	768 (0.38)	2.6	7.3	22.4	67.7
64	2400	520	1308 (0.65)	3.1	4.3	52.9	39.8	788 (0.39)	5.1	7.1	21.8	66.0
64	4800	520	1348 (0.67)	5.9	4.2	51.4	38.6	828 (0.41)	9.7	6.8	20.8	62.8
64	9600	520	1428 (0.71)	11.2	3.9	48.5	36.4	908 (0.45)	17.6	6.2	18.9	57.3

B.3.3. SPUR Floating-point Performance

Table B-13. Simulation Results for SPUR Running GE COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	56	180 (0.57)	0.0	10.6	58.3	31.1	144 (0.45)	0.0	10.4	50.7	38.9
8	300	56	186 (0.58)	3.1	10.2	56.6	30.1	150 (0.47)	2.0	10.0	50.6	37.3
8	600	56	190 (0.60)	5.1	10.0	55.5	29.5	154 (0.48)	3.3	9.7	50.7	36.4
8	1200	56	198 (0.62)	9.0	9.6	53.1	28.3	162 (0.51)	7.1	9.3	49.0	34.6
8	2400	56	216 (0.68)	16.6	8.8	48.7	25.9	180 (0.57)	16.4	8.3	44.2	31.1
8	4800	56	250 (0.79)	27.9	7.6	42.1	22.4	214 (0.67)	29.7	7.0	37.1	26.2
8	9600	56	318 (1.00)	43.3	6.0	33.1	17.6	282 (0.89)	46.6	5.3	28.2	19.9
16	0	56	132 (0.42)	0.0	14.4	43.2	42.4	100 (0.31)	0.0	10.0	34.0	56.0
16	300	56	138 (0.43)	4.2	13.8	41.6	40.6	105 (0.33)	2.9	10.5	33.3	53.3
16	600	56	142 (0.45)	6.9	13.4	40.3	39.4	109 (0.34)	4.6	10.3	33.8	51.4
16	1200	56	150 (0.47)	11.8	12.7	38.1	37.3	117 (0.37)	9.8	9.6	32.7	47.9
16	2400	56	168 (0.53)	21.3	11.3	34.1	33.3	135 (0.42)	21.9	8.3	28.3	41.5
16	4800	56	202 (0.64)	34.5	9.4	28.4	27.7	169 (0.53)	37.6	6.7	22.6	33.1
16	9600	56	270 (0.85)	51.0	7.0	21.1	20.7	237 (0.75)	55.5	4.8	16.1	23.6
32	0	56	108 (0.34)	0.0	17.6	30.6	51.9	84 (0.26)	0.0	11.9	21.4	66.7
32	300	56	114 (0.36)	5.0	16.7	29.2	49.1	87 (0.27)	3.5	11.5	20.7	64.4
32	600	56	118 (0.37)	8.3	16.1	28.2	47.5	91 (0.29)	5.5	12.1	20.9	61.5
32	1200	56	126 (0.40)	14.1	15.1	26.4	44.4	99 (0.31)	11.6	11.4	20.5	56.6
32	2400	56	144 (0.45)	24.8	13.2	23.1	38.9	117 (0.37)	25.2	9.6	17.3	47.9
32	4800	56	178 (0.56)	39.2	10.7	18.7	31.5	151 (0.47)	42.1	7.5	13.4	37.1
32	9600	56	246 (0.77)	56.0	7.7	13.5	22.8	219 (0.69)	60.0	5.1	9.3	25.6
64	0	56	96 (0.30)	0.0	19.8	21.9	58.3	76 (0.24)	0.0	13.2	13.1	73.7
64	300	56	102 (0.32)	5.6	18.6	20.9	54.9	79 (0.25)	3.8	12.7	12.7	70.9
64	600	56	106 (0.33)	9.2	17.9	20.1	52.8	82 (0.26)	6.1	12.8	12.8	68.3
64	1200	56	114 (0.36)	15.6	16.7	18.7	49.1	90 (0.28)	12.8	12.5	12.5	62.2
64	2400	56	132 (0.42)	27.1	14.4	16.1	42.4	108 (0.34)	27.3	10.4	10.5	51.9
64	4800	56	166 (0.52)	42.0	11.4	12.8	33.7	142 (0.45)	44.7	7.9	7.9	39.4
64	9600	56	234 (0.74)	58.9	8.1	9.2	23.9	210 (0.66)	62.6	5.4	5.4	26.7

Table B-14. Simulation Results for SPUR Running GE ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	56	168 (0.55)	0.0	6.6	60.2	33.3	132 (0.43)	0.0	8.3	49.2	42.4
8	300	56	174 (0.57)	3.2	6.3	58.4	32.2	138 (0.45)	1.1	8.0	50.4	40.6
8	600	56	178 (0.58)	5.3	6.2	57.0	31.5	142 (0.46)	2.5	7.8	50.3	39.4
8	1200	56	186 (0.61)	9.4	5.9	54.6	30.1	150 (0.49)	6.7	7.3	48.6	37.3
8	2400	56	204 (0.67)	17.4	5.4	49.7	27.5	168 (0.55)	16.7	6.6	43.5	33.3
8	4800	56	238 (0.78)	29.2	4.6	42.6	23.5	202 (0.66)	30.7	5.5	36.1	27.7
8	9600	56	306 (1.00)	44.9	3.6	33.3	18.3	270 (0.88)	48.1	4.1	27.0	20.7
16	0	56	120 (0.39)	0.0	9.2	44.2	46.7	88 (0.29)	0.0	6.8	29.5	63.6
16	300	56	126 (0.41)	4.4	8.7	42.5	44.4	93 (0.30)	1.6	7.5	30.6	60.2
16	600	56	130 (0.42)	7.3	8.5	41.1	43.1	97 (0.32)	3.6	7.5	31.2	57.7
16	1200	56	138 (0.45)	12.7	8.0	38.9	40.6	105 (0.34)	9.5	6.9	30.2	53.3
16	2400	56	156 (0.51)	22.8	7.1	34.4	35.9	123 (0.40)	22.8	5.9	25.8	45.5
16	4800	56	190 (0.62)	36.6	5.8	28.2	29.5	157 (0.51)	39.5	4.6	20.2	35.7
16	9600	56	258 (0.84)	53.3	4.3	20.7	21.7	225 (0.74)	57.8	3.2	14.2	24.9
32	0	56	96 (0.31)	0.0	11.5	30.3	58.3	72 (0.24)	0.0	6.9	15.3	77.8
32	300	56	102 (0.33)	5.4	10.8	29.0	54.9	75 (0.25)	2.0	7.0	16.3	74.7
32	600	56	106 (0.35)	9.0	10.4	27.9	52.8	79 (0.26)	4.4	7.9	16.8	70.9
32	1200	56	114 (0.37)	15.4	9.7	26.0	49.1	87 (0.28)	11.5	7.5	16.6	64.4
32	2400	56	132 (0.43)	26.9	8.3	22.4	42.4	105 (0.34)	26.7	6.2	13.8	53.3
32	4800	56	166 (0.54)	41.9	6.6	17.8	33.7	139 (0.45)	44.6	4.7	10.4	40.3
32	9600	56	234 (0.76)	58.8	4.7	12.7	23.9	207 (0.68)	62.8	3.1	7.0	27.1
64	0	56	84 (0.27)	0.0	13.1	20.2	66.7	64 (0.21)	0.0	6.3	6.3	87.5
64	300	56	90 (0.29)	6.1	12.2	19.5	62.2	67 (0.22)	2.2	6.7	7.5	83.6
64	600	56	94 (0.31)	10.1	11.7	18.6	59.6	70 (0.23)	5.0	7.1	7.9	80.0
64	1200	56	102 (0.33)	17.2	10.8	17.2	54.9	78 (0.25)	12.8	7.4	8.0	71.8
64	2400	56	120 (0.39)	29.6	9.2	14.6	46.7	96 (0.31)	29.2	6.0	6.5	58.3
64	4800	56	154 (0.50)	45.1	7.1	11.4	36.4	130 (0.42)	47.7	4.4	4.9	43.1
64	9600	56	222 (0.73)	61.9	5.0	7.9	25.2	198 (0.65)	65.7	2.9	3.1	28.3

Table B-15. Simulation Results for SPUR Running DP COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	44	132 (0.49)	0.0	10.6	56.1	33.3	124 (0.46)	0.0	8.1	56.4	35.5
8	300	44	138 (0.51)	4.4	10.1	53.6	31.9	130 (0.48)	4.4	7.7	54.1	33.8
8	600	44	142 (0.53)	7.0	9.9	52.1	31.0	134 (0.50)	7.3	7.5	52.4	32.8
8	1200	44	150 (0.56)	12.0	9.3	49.4	29.3	142 (0.53)	12.5	7.0	49.5	31.0
8	2400	44	168 (0.62)	21.4	8.3	44.1	26.2	160 (0.59)	22.3	6.3	43.9	27.5
8	4800	44	202 (0.75)	34.7	6.9	36.7	21.8	194 (0.72)	36.0	5.2	36.2	22.7
8	9600	44	270 (1.00)	51.1	5.2	27.4	16.3	262 (0.97)	52.6	3.8	26.8	16.8
16	0	44	100 (0.37)	0.0	14.0	42.0	44.0	92 (0.34)	0.0	10.9	41.3	47.8
16	300	44	106 (0.39)	5.7	13.2	39.5	41.5	98 (0.36)	5.9	10.2	39.0	44.9
16	600	44	110 (0.41)	9.1	12.7	38.1	40.0	102 (0.38)	9.6	9.8	37.5	43.1
16	1200	44	118 (0.44)	15.3	11.9	35.5	37.3	110 (0.41)	16.1	9.1	34.8	40.0
16	2400	44	136 (0.50)	26.5	10.3	30.8	32.4	128 (0.47)	27.9	7.8	29.9	34.4
16	4800	44	170 (0.63)	41.2	8.2	24.8	25.9	162 (0.60)	43.1	6.2	23.6	27.2
16	9600	44	238 (0.88)	58.0	5.9	17.7	18.5	230 (0.85)	59.9	4.4	16.6	19.1
32	0	44	84 (0.31)	0.0	16.7	31.0	52.4	76 (0.28)	0.0	13.2	28.9	57.9
32	300	44	90 (0.33)	6.7	15.6	28.8	48.9	82 (0.30)	7.0	12.2	27.1	53.7
32	600	44	94 (0.35)	10.6	14.9	27.7	46.8	86 (0.32)	11.3	11.6	25.9	51.2
32	1200	44	102 (0.38)	17.6	13.7	25.5	43.1	94 (0.35)	18.9	10.6	23.7	46.8
32	2400	44	120 (0.44)	30.0	11.7	21.6	36.7	112 (0.41)	31.9	8.9	19.9	39.3
32	4800	44	154 (0.57)	45.5	9.1	16.9	28.6	146 (0.54)	47.8	6.9	15.3	30.1
32	9600	44	222 (0.82)	62.2	6.3	11.8	19.8	214 (0.79)	64.4	4.7	10.4	20.6
64	0	44	76 (0.28)	0.0	18.4	23.7	57.9	68 (0.25)	0.0	14.7	20.6	64.7
64	300	44	82 (0.30)	7.3	17.1	22.0	53.7	74 (0.27)	7.8	13.5	19.3	59.5
64	600	44	86 (0.32)	11.6	16.3	21.0	51.2	78 (0.29)	12.5	12.8	18.3	56.4
64	1200	44	94 (0.35)	19.1	14.9	19.2	46.8	86 (0.32)	20.6	11.6	16.6	51.2
64	2400	44	112 (0.41)	32.1	12.5	16.1	39.3	104 (0.39)	34.4	9.6	13.7	42.3
64	4800	44	146 (0.54)	47.9	9.6	12.4	30.1	138 (0.51)	50.5	7.3	10.3	31.9
64	9600	44	214 (0.79)	64.5	6.5	8.4	20.6	206 (0.76)	66.9	4.9	6.9	21.4

Table B-16. Simulation Results for SPUR Running DP ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	44	132 (0.49)	0.0	3.8	62.9	33.3	114 (0.42)	0.0	3.5	57.9	38.6
8	300	44	138 (0.51)	4.4	3.6	60.1	31.9	120 (0.44)	5.0	3.3	55.0	36.7
8	600	44	142 (0.53)	7.0	3.5	58.5	31.0	124 (0.46)	8.1	3.2	53.2	35.5
8	1200	44	150 (0.56)	12.0	3.3	55.3	29.3	132 (0.49)	13.6	3.0	50.0	33.3
8	2400	44	168 (0.62)	21.4	3.0	49.4	26.2	150 (0.56)	24.0	2.7	44.0	29.3
8	4800	44	202 (0.75)	34.7	2.5	41.0	21.8	184 (0.68)	38.0	2.2	35.9	23.9
8	9600	44	270 (1.00)	51.1	1.9	30.8	16.3	252 (0.93)	54.8	1.6	26.2	17.5
16	0	44	92 (0.34)	0.0	5.4	46.7	47.8	74 (0.27)	0.0	5.4	35.2	59.5
16	300	44	98 (0.36)	6.1	5.1	43.9	44.9	80 (0.30)	7.5	5.0	32.5	55.0
16	600	44	102 (0.38)	9.8	4.9	42.2	43.1	84 (0.31)	11.9	4.8	31.0	52.4
16	1200	44	110 (0.41)	16.4	4.6	39.1	40.0	92 (0.34)	19.6	4.4	28.2	47.8
16	2400	44	128 (0.47)	28.1	3.9	33.6	34.4	110 (0.41)	32.7	3.6	23.7	40.0
16	4800	44	162 (0.60)	43.2	3.1	26.6	27.2	144 (0.53)	48.6	2.8	18.0	30.6
16	9600	44	230 (0.85)	60.0	2.2	18.7	19.1	212 (0.79)	65.1	1.9	12.2	20.8
32	0	44	72 (0.27)	0.0	6.9	32.0	61.1	54 (0.20)	0.0	0.9	17.6	81.5
32	300	44	78 (0.29)	7.7	6.4	29.5	56.4	60 (0.22)	8.8	0.8	17.1	73.3
32	600	44	82 (0.30)	12.2	6.1	28.1	53.7	64 (0.24)	14.5	0.8	16.0	68.8
32	1200	44	90 (0.33)	20.0	5.6	25.6	48.9	72 (0.27)	24.0	0.7	14.2	61.1
32	2400	44	108 (0.40)	33.3	4.6	21.3	40.7	90 (0.33)	39.2	0.6	11.4	48.9
32	4800	44	142 (0.53)	49.3	3.5	16.3	31.0	124 (0.46)	55.8	0.4	8.3	35.5
32	9600	44	210 (0.78)	65.7	2.4	11.0	21.0	192 (0.71)	71.5	0.3	5.4	22.9
64	0	44	62 (0.23)	0.0	8.1	21.0	71.0	44 (0.16)	0.0	0.0	0.0	100.0
64	300	44	68 (0.25)	8.5	7.4	19.5	64.7	50 (0.19)	5.5	0.0	6.5	88.0
64	600	44	72 (0.27)	13.5	6.9	18.5	61.1	54 (0.20)	12.0	0.0	6.5	81.5
64	1200	44	80 (0.30)	22.2	6.3	16.6	55.0	62 (0.23)	23.4	0.0	5.6	71.0
64	2400	44	98 (0.36)	36.5	5.1	13.5	44.9	80 (0.30)	40.6	0.0	4.4	55.0
64	4800	44	132 (0.49)	52.8	3.8	10.1	33.3	114 (0.42)	58.3	0.0	3.0	38.6
64	9600	44	200 (0.74)	68.9	2.5	6.7	22.0	182 (0.67)	73.9	0.0	2.0	24.2

Table B-17. Simulation Results for SPUR Running PE COMP

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	44	92 (0.57)	0.0	10.9	41.4	47.8	64 (0.40)	0.0	4.7	26.6	68.8
8	300	44	95 (0.59)	3.2	10.5	40.1	46.3	67 (0.42)	0.0	4.5	29.9	65.7
8	600	44	97 (0.60)	5.2	10.3	39.2	45.4	69 (0.43)	1.1	4.4	30.8	63.8
8	1200	44	101 (0.63)	8.9	9.9	37.7	43.6	73 (0.45)	6.2	4.1	29.5	60.3
8	2400	44	110 (0.68)	16.4	9.1	34.5	40.0	82 (0.51)	16.5	3.7	26.2	53.7
8	4800	44	127 (0.79)	27.6	7.9	30.0	34.6	99 (0.61)	30.8	3.0	21.7	44.4
8	9600	44	161 (1.00)	42.9	6.2	23.6	27.3	133 (0.83)	48.5	2.3	16.2	33.1
16	0	44	76 (0.47)	0.0	13.2	29.0	57.9	48 (0.30)	0.0	2.1	6.3	91.7
16	300	44	79 (0.49)	3.8	12.7	27.9	55.7	51 (0.32)	0.0	2.9	10.8	86.3
16	600	44	81 (0.50)	6.2	12.3	27.1	54.3	53 (0.33)	1.4	2.8	12.7	83.0
16	1200	44	85 (0.53)	10.6	11.8	25.9	51.8	57 (0.35)	7.9	2.6	12.3	77.2
16	2400	44	94 (0.58)	19.1	10.6	23.4	46.8	66 (0.41)	20.5	2.3	10.6	66.7
16	4800	44	111 (0.69)	31.5	9.0	19.8	39.6	83 (0.52)	36.7	1.8	8.4	53.0
16	9600	44	145 (0.90)	47.6	6.9	15.2	30.3	117 (0.73)	55.1	1.3	6.0	37.6
32	0	44	68 (0.42)	0.0	14.7	20.6	64.7	44 (0.27)	0.0	0.0	0.0	100.0
32	300	44	71 (0.44)	4.2	14.1	19.7	62.0	46 (0.29)	0.0	1.1	3.2	95.7
32	600	44	73 (0.45)	6.9	13.7	19.2	60.3	48 (0.30)	1.6	1.6	5.2	91.7
32	1200	44	77 (0.48)	11.7	13.0	18.2	57.1	52 (0.32)	8.7	1.4	5.3	84.6
32	2400	44	86 (0.53)	20.9	11.6	16.3	51.2	61 (0.38)	22.1	1.2	4.5	72.1
32	4800	44	103 (0.64)	34.0	9.7	13.6	42.7	78 (0.48)	39.1	1.0	3.5	56.4
32	9600	44	137 (0.85)	50.4	7.3	10.2	32.1	112 (0.70)	57.6	0.7	2.4	39.3
64	0	44	64 (0.40)	0.0	15.6	15.7	68.8	44 (0.27)	0.0	0.0	0.0	100.0
64	300	44	67 (0.42)	4.5	14.9	15.0	65.7	45 (0.28)	0.0	0.6	1.7	97.8
64	600	44	69 (0.43)	7.3	14.5	14.6	63.8	47 (0.29)	1.6	1.6	3.2	93.6
64	1200	44	73 (0.45)	12.3	13.7	13.7	60.3	51 (0.32)	8.8	1.5	3.4	86.3
64	2400	44	82 (0.51)	22.0	12.2	12.2	53.7	60 (0.37)	22.5	1.3	2.9	73.3
64	4800	44	99 (0.61)	35.4	10.1	10.1	44.4	77 (0.48)	39.6	1.0	2.3	57.1
64	9600	44	133 (0.83)	51.9	7.5	7.6	33.1	111 (0.69)	58.1	0.7	1.6	39.6

Table B-18. Simulation Results for SPUR Running PE ASSM

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	44	84 (0.55)	0.0	3.6	44.1	52.4	52 (0.34)	0.0	3.9	11.5	84.6
8	300	44	87 (0.57)	3.2	3.5	42.9	50.6	55 (0.36)	0.0	3.6	16.3	80.0
8	600	44	89 (0.58)	5.3	3.4	41.9	49.4	57 (0.37)	0.0	3.5	19.3	77.2
8	1200	44	93 (0.61)	9.4	3.2	40.1	47.3	61 (0.40)	4.1	3.3	20.5	72.1
8	2400	44	102 (0.67)	17.4	2.9	36.6	43.1	70 (0.46)	16.4	2.9	17.8	62.9
8	4800	44	119 (0.78)	29.2	2.5	31.3	37.0	87 (0.57)	32.8	2.3	14.4	50.6
8	9600	44	153 (1.00)	44.9	2.0	24.4	28.8	121 (0.79)	51.7	1.7	10.3	36.4
16	0	44	68 (0.44)	0.0	4.4	30.9	64.7	44 (0.29)	0.0	0.0	0.0	100.0
16	300	44	71 (0.46)	3.9	4.2	29.9	62.0	45 (0.29)	0.0	0.6	1.7	97.8
16	600	44	73 (0.48)	6.5	4.1	29.1	60.3	47 (0.31)	0.0	1.1	5.3	93.6
16	1200	44	77 (0.50)	11.4	3.9	27.6	57.1	51 (0.33)	4.9	1.0	7.9	86.3
16	2400	44	86 (0.56)	20.6	3.5	24.7	51.2	60 (0.39)	19.2	0.8	6.7	73.3
16	4800	44	103 (0.67)	33.7	2.9	20.6	42.7	77 (0.50)	37.0	0.6	5.2	57.1
16	9600	44	137 (0.90)	50.2	2.2	15.6	32.1	111 (0.73)	56.3	0.5	3.6	39.6
32	0	44	60 (0.39)	0.0	5.0	21.6	73.3	44 (0.29)	0.0	0.0	0.0	100.0
32	300	44	63 (0.41)	4.4	4.8	21.0	69.8	44 (0.29)	0.0	0.0	0.0	100.0
32	600	44	65 (0.42)	7.3	4.6	20.4	67.7	45 (0.29)	0.0	0.6	1.7	97.8
32	1200	44	69 (0.45)	12.7	4.4	19.3	63.8	49 (0.32)	5.1	1.0	4.1	89.8
32	2400	44	78 (0.51)	22.8	3.9	17.0	56.4	58 (0.38)	19.8	0.9	3.4	75.9
32	4800	44	95 (0.62)	36.6	3.2	14.0	46.3	75 (0.49)	38.0	0.7	2.6	58.7
32	9600	44	129 (0.84)	53.3	2.3	10.3	34.1	109 (0.71)	57.3	0.5	1.8	40.4
64	0	44	56 (0.37)	0.0	5.4	16.1	78.6	44 (0.29)	0.0	0.0	0.0	100.0
64	300	44	59 (0.39)	4.7	5.1	15.7	74.6	44 (0.29)	0.0	0.0	0.0	100.0
64	600	44	61 (0.40)	7.8	4.9	15.2	72.1	44 (0.29)	0.0	0.0	0.0	100.0
64	1200	44	65 (0.42)	13.5	4.6	14.3	67.7	48 (0.31)	5.2	1.0	2.1	91.7
64	2400	44	74 (0.48)	24.0	4.1	12.5	59.5	57 (0.37)	20.2	0.9	1.7	77.2
64	4800	44	91 (0.59)	38.2	3.3	10.2	48.4	74 (0.48)	38.5	0.7	1.3	59.5
64	9600	44	125 (0.82)	55.0	2.4	7.4	35.2	108 (0.71)	57.9	0.5	0.9	40.7

B.3.4. Intel Floating-point Performance — Non-Concurrent Model

Table B-19. Simulation Results for Intel Running DP COMP Non-parallel

Bus Width To FPU (bits)	Cache Time per Miss (nsec)	Tot FPU Cycles	Sequential Execution					Parallel Execution				
			Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op	Tot Prog Cycles	% Cache OvHd	% Loop OvHd	% FP OvHd	% FP Op
8	0	2388	5224 (0.96)	0.0	5.7	48.6	45.7	5224 (0.96)	0.0	5.7	48.6	45.7
8	300	2388	5230 (0.97)	0.1	5.7	48.6	45.7	5230 (0.97)	0.1	5.7	48.6	45.7
8	600	2388	5236 (0.97)	0.2	5.7	48.5	45.6	5236 (0.97)	0.2	5.7	48.5	45.6
8	1200	2388	5248 (0.97)	0.5	5.6	48.4	45.5	5248 (0.97)	0.5	5.6	48.4	45.5
8	2400	2388	5272 (0.97)	0.9	5.6	48.2	45.3	5272 (0.97)	0.9	5.6	48.2	45.3
8	4800	2388	5320 (0.98)	1.8	5.6	47.7	44.9	5320 (0.98)	1.8	5.6	47.7	44.9
8	9600	2388	5416 (1.00)	3.6	5.5	46.9	44.1	5416 (1.00)	3.6	5.5	46.9	44.1
16	0	2388	4456 (0.82)	0.0	6.6	39.8	53.6	4456 (0.82)	0.0	6.6	39.8	53.6
16	300	2388	4462 (0.82)	0.1	6.6	39.7	53.5	4462 (0.82)	0.1	6.6	39.7	53.5
16	600	2388	4468 (0.82)	0.3	6.6	39.7	53.4	4468 (0.82)	0.3	6.6	39.7	53.4
16	1200	2388	4480 (0.83)	0.5	6.6	39.6	53.3	4480 (0.83)	0.5	6.6	39.6	53.3
16	2400	2388	4504 (0.83)	1.1	6.6	39.3	53.0	4504 (0.83)	1.1	6.6	39.3	53.0
16	4800	2388	4552 (0.84)	2.1	6.5	38.9	52.5	4552 (0.84)	2.1	6.5	38.9	52.5
16	9600	2388	4648 (0.86)	4.1	6.4	38.1	51.4	4648 (0.86)	4.1	6.4	38.1	51.4
32	0	2388	4072 (0.75)	0.0	7.3	34.1	58.6	4072 (0.75)	0.0	7.3	34.1	58.6
32	300	2388	4078 (0.75)	0.1	7.3	34.0	58.6	4078 (0.75)	0.1	7.3	34.0	58.6
32	600	2388	4084 (0.75)	0.3	7.3	34.0	58.5	4084 (0.75)	0.3	7.3	34.0	58.5
32	1200	2388	4096 (0.76)	0.6	7.2	33.9	58.3	4096 (0.76)	0.6	7.2	33.9	58.3
32	2400	2388	4120 (0.76)	1.2	7.2	33.7	58.0	4120 (0.76)	1.2	7.2	33.7	58.0
32	4800	2388	4168 (0.77)	2.3	7.1	33.3	57.3	4168 (0.77)	2.3	7.1	33.3	57.3
32	9600	2388	4264 (0.79)	4.5	6.9	32.6	56.0	4264 (0.79)	4.5	6.9	32.6	56.0
64	0	2388	4136 (0.76)	0.0	7.2	35.1	57.7	4136 (0.76)	0.0	7.2	35.1	57.7
64	300	2388	4142 (0.76)	0.1	7.2	35.1	57.7	4142 (0.76)	0.1	7.2	35.1	57.7
64	600	2388	4148 (0.77)	0.3	7.1	35.0	57.6	4148 (0.77)	0.3	7.1	35.0	57.6
64	1200	2388	4160 (0.77)	0.6	7.1	34.9	57.4	4160 (0.77)	0.6	7.1	34.9	57.4
64	2400	2388	4184 (0.77)	1.2	7.1	34.7	57.1	4184 (0.77)	1.2	7.1	34.7	57.1
64	4800	2388	4232 (0.78)	2.3	7.0	34.3	56.4	4232 (0.78)	2.3	7.0	34.3	56.4
64	9600	2388	4328 (0.80)	4.4	6.8	33.5	55.2	4328 (0.80)	4.4	6.8	33.5	55.2

B.4. Floating-point Performance Histograms

The data contained in Table B-1 through Table B-18 are illustrated in Figure B-1 through Figure B-9. The floating-point operation time, floating-point overhead, loop overhead, and cache-miss overhead are shown for different size busses and all versions of all programs. The left-most histogram bar of each figure represents the longest execution time, and all others are normalized to it. The left-most histogram-bar of each pair shows the execution time for sequential operation (i,e., no concurrency between the CPU and FPU) and the right-most bar shows parallel operation for the same cache service time and buswidth. The number above the left-most bar in each pair is the cache service time in nanoseconds divided by 100.

B.4.1. Intel Floating-point Performance Histograms

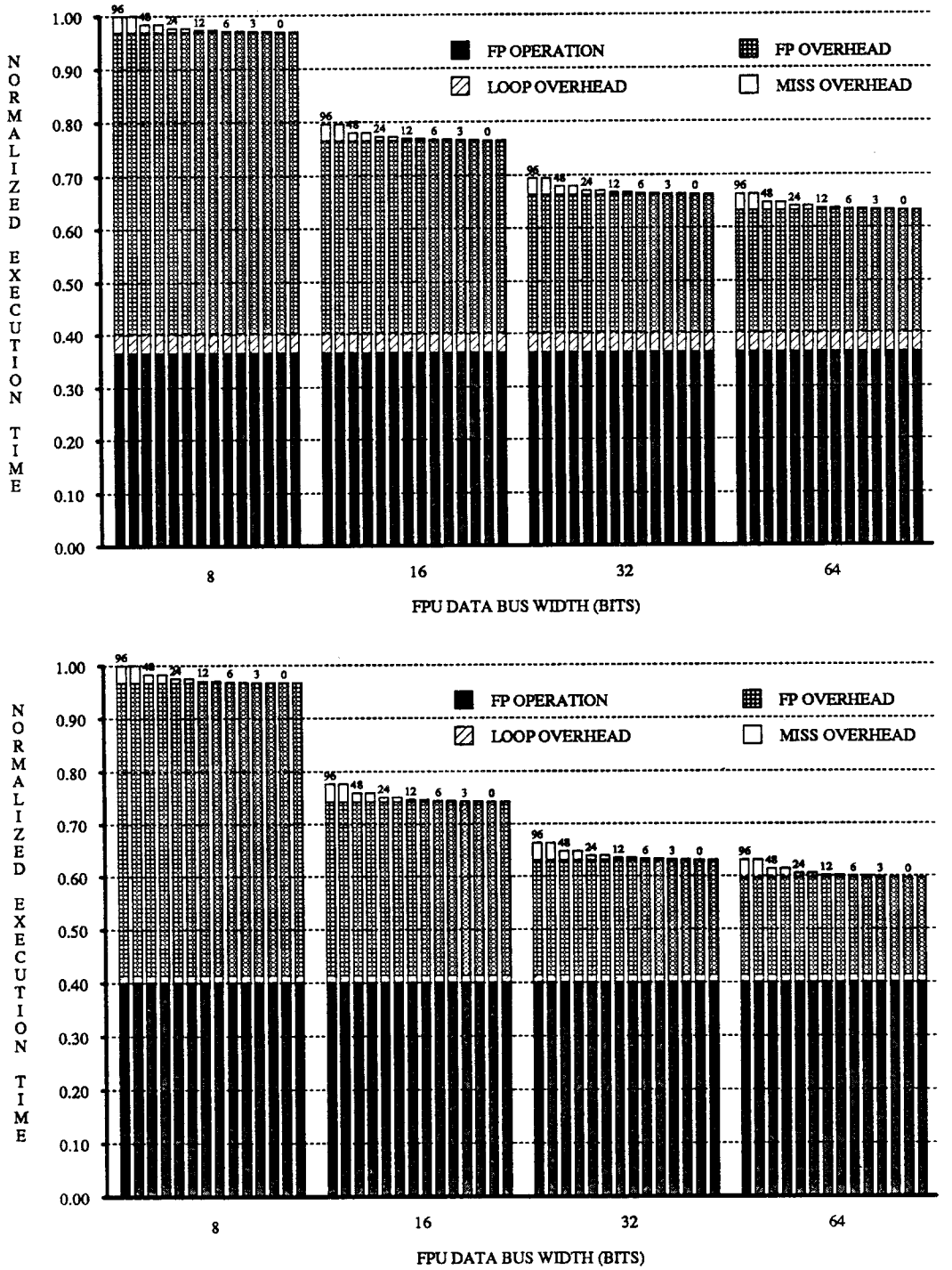


Figure B-1. Intel System Performance for GE Programs.

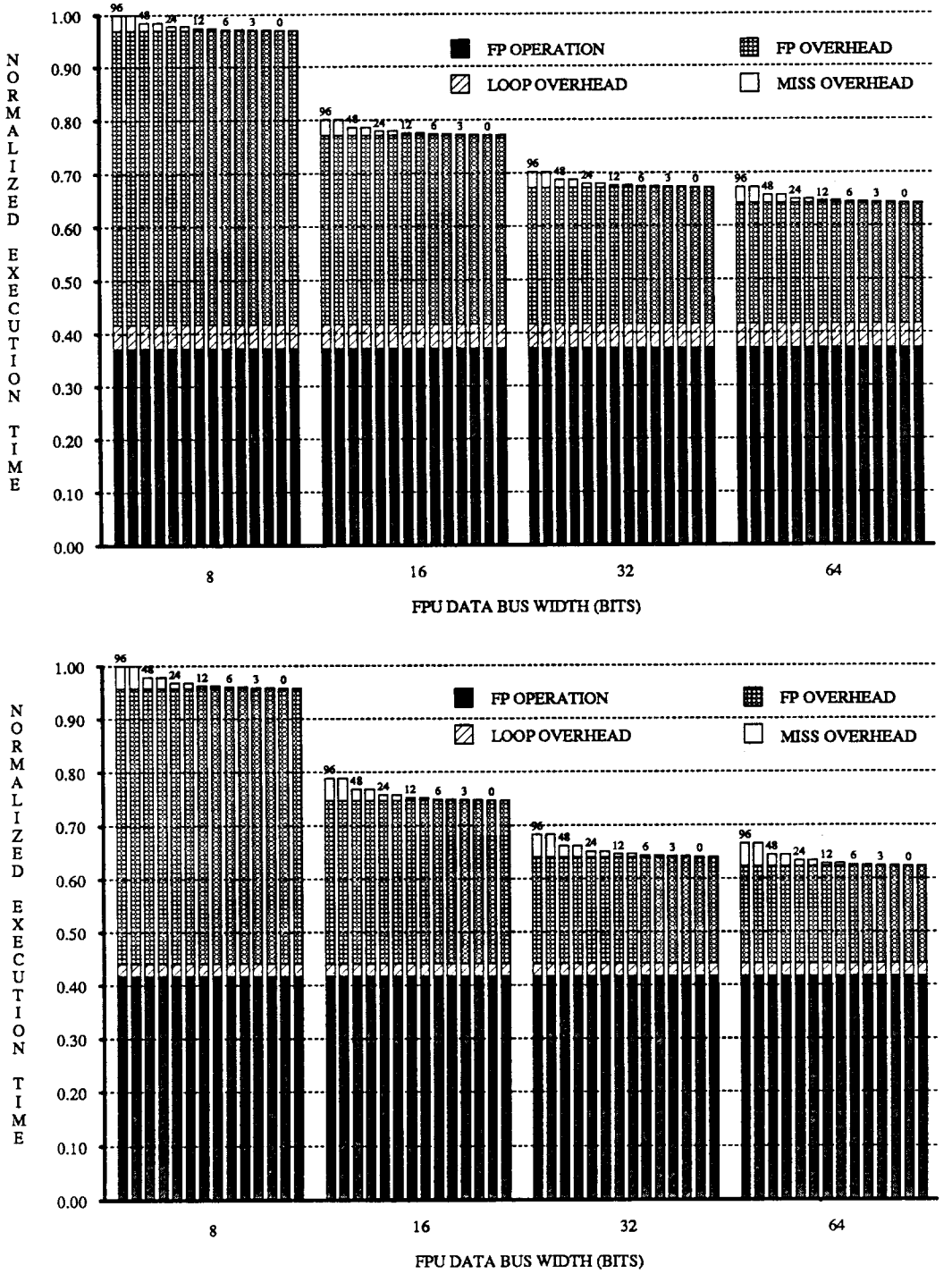


Figure B-2. Intel System Performance for DP Programs.

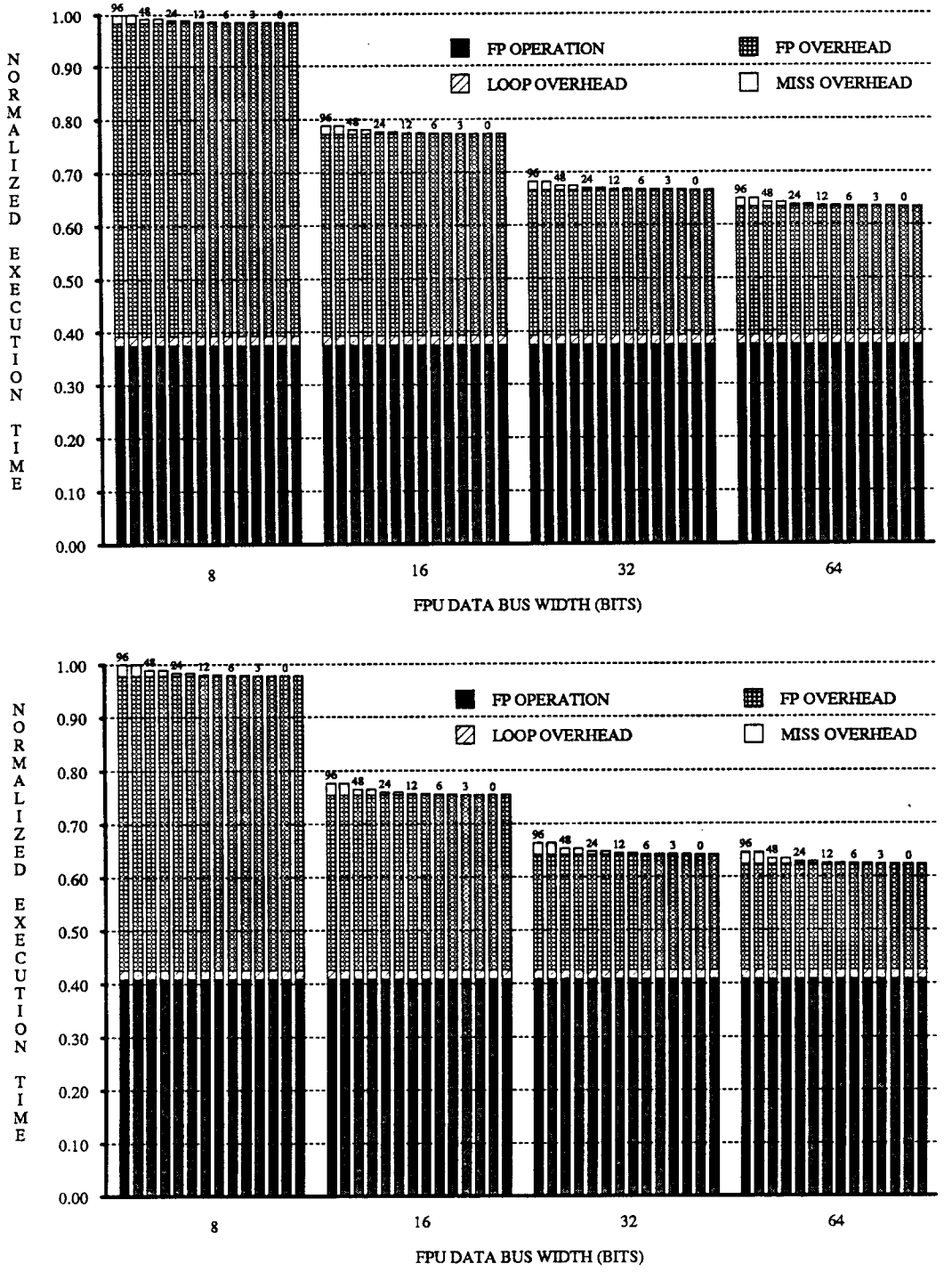


Figure B-3. Intel System Performance for PE Programs.

B.4.2. Motorola Floating-point Performance Histograms

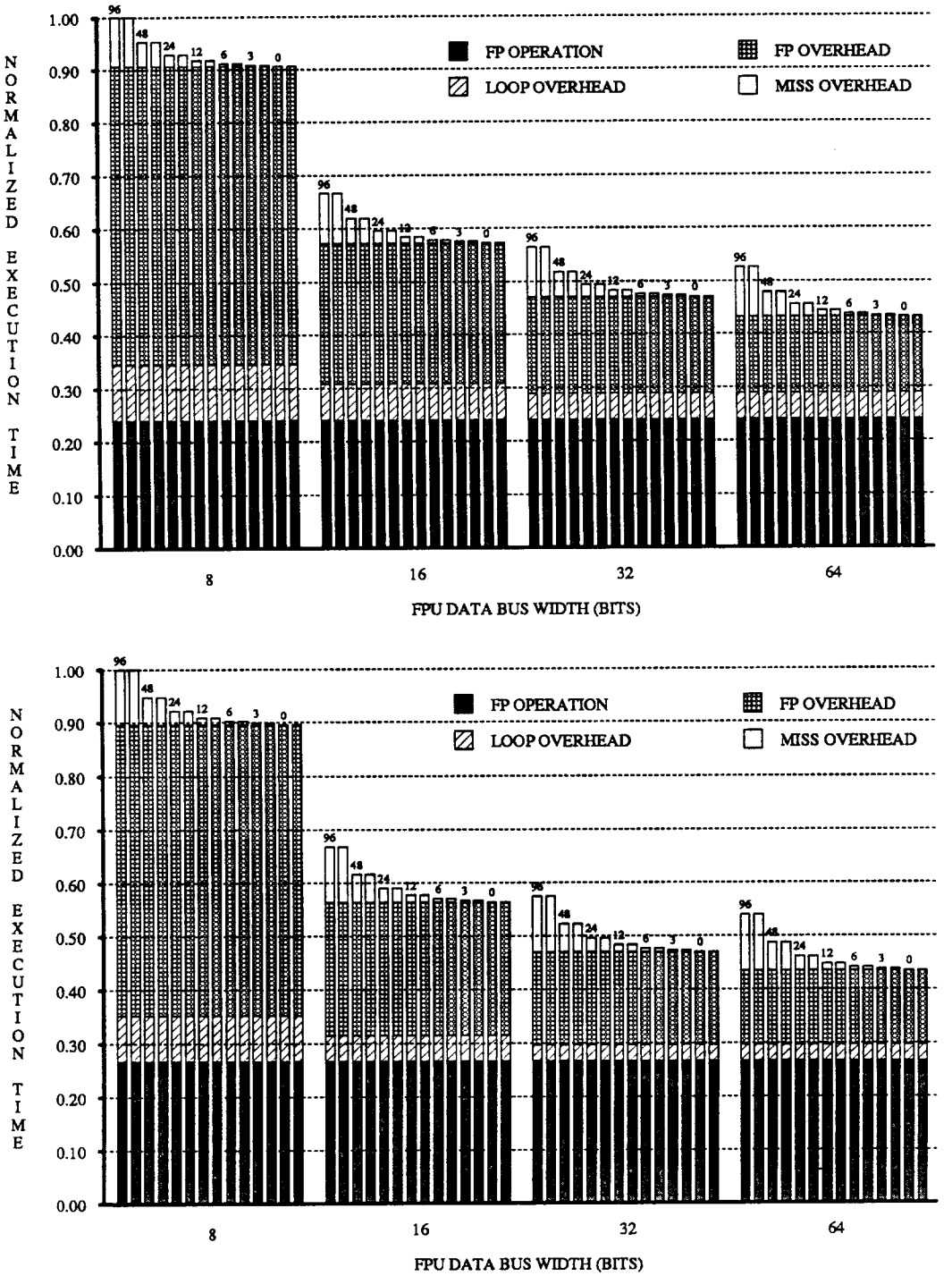


Figure B-4. Motorola System Performance for GE Programs.

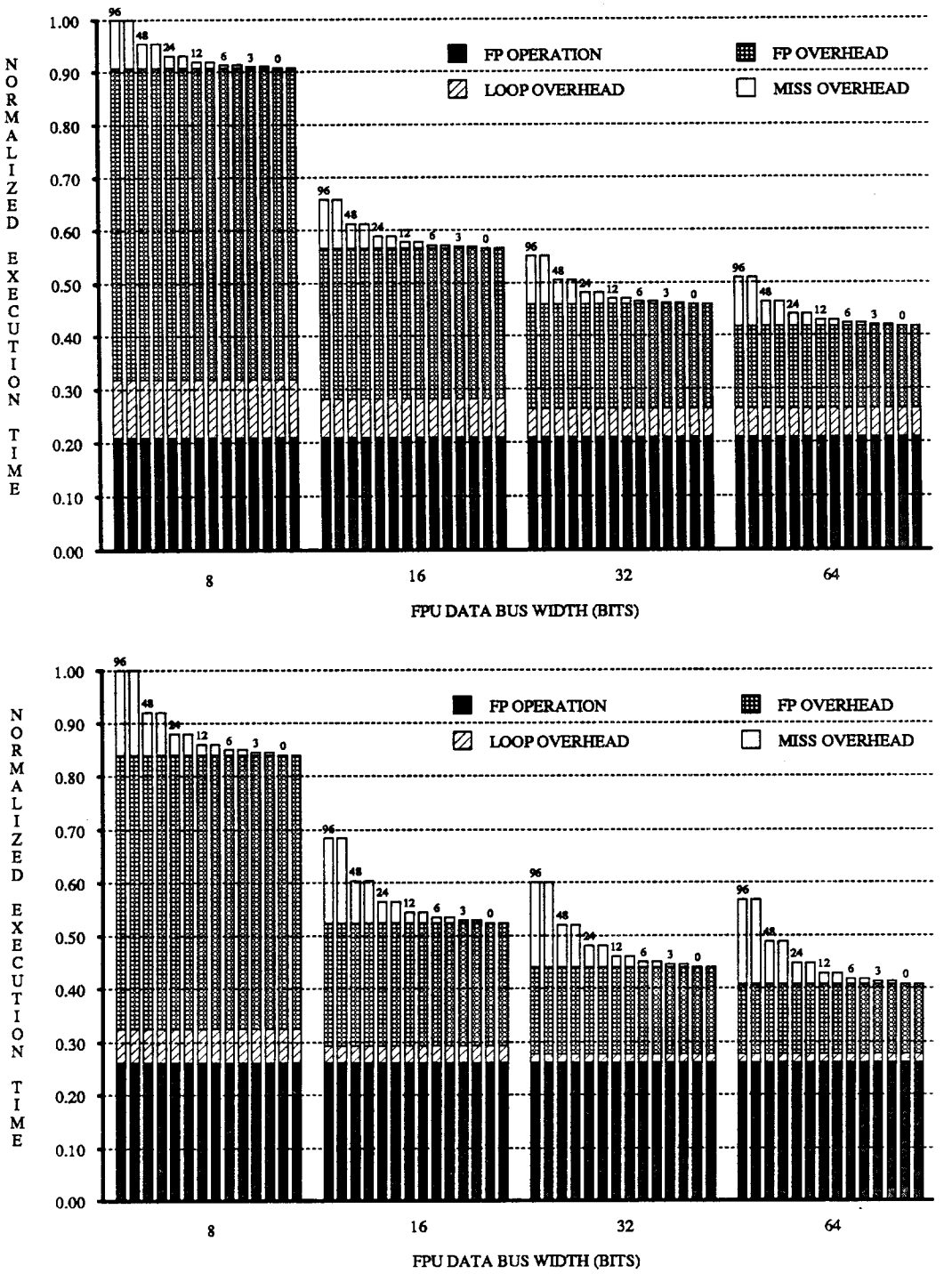


Figure B-5. Motorola System Performance for DP Programs.

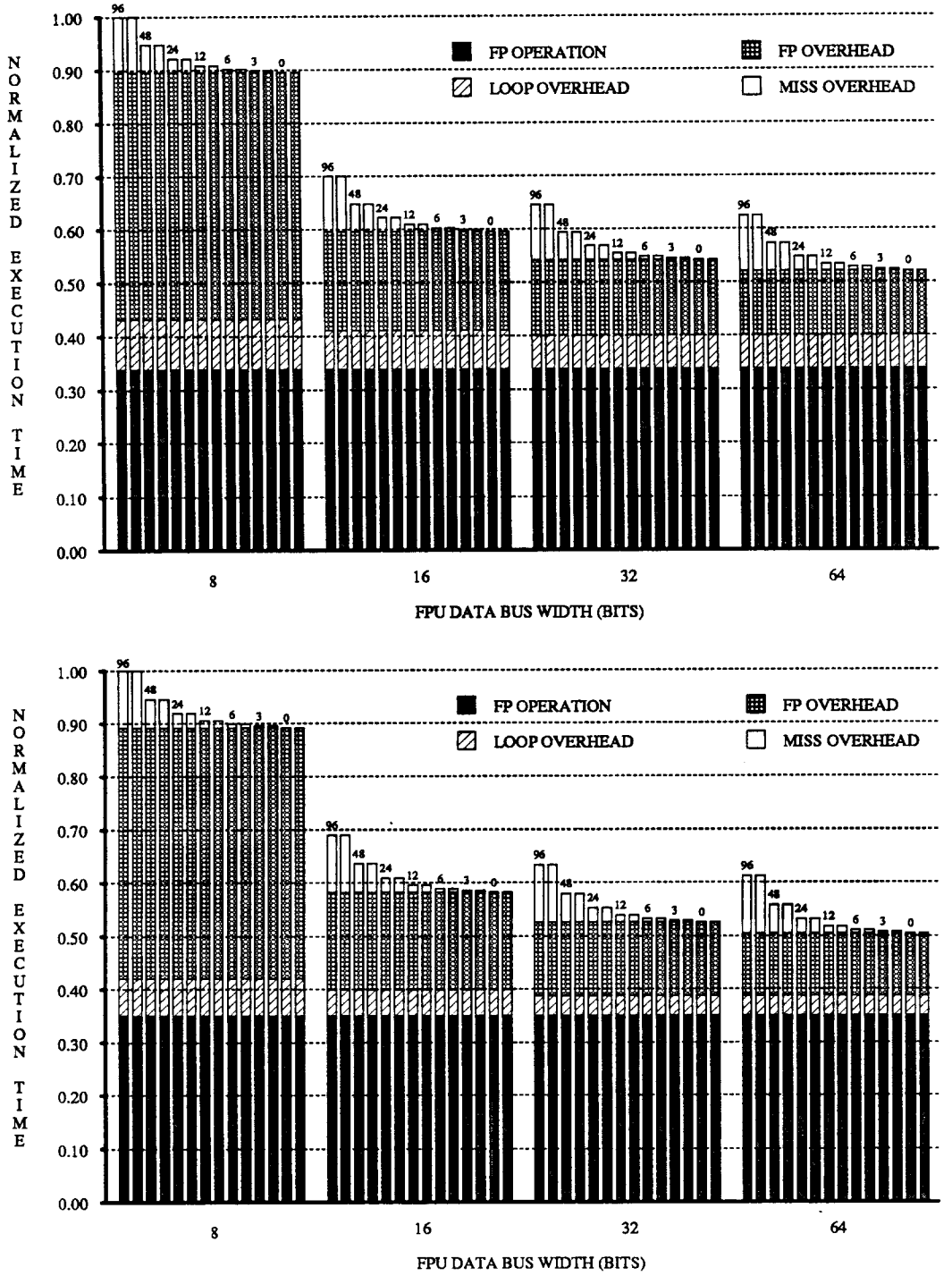


Figure B-6. Motorola System Performance for PE Programs.

B.4.3. SPUR Floating-point Performance Histograms

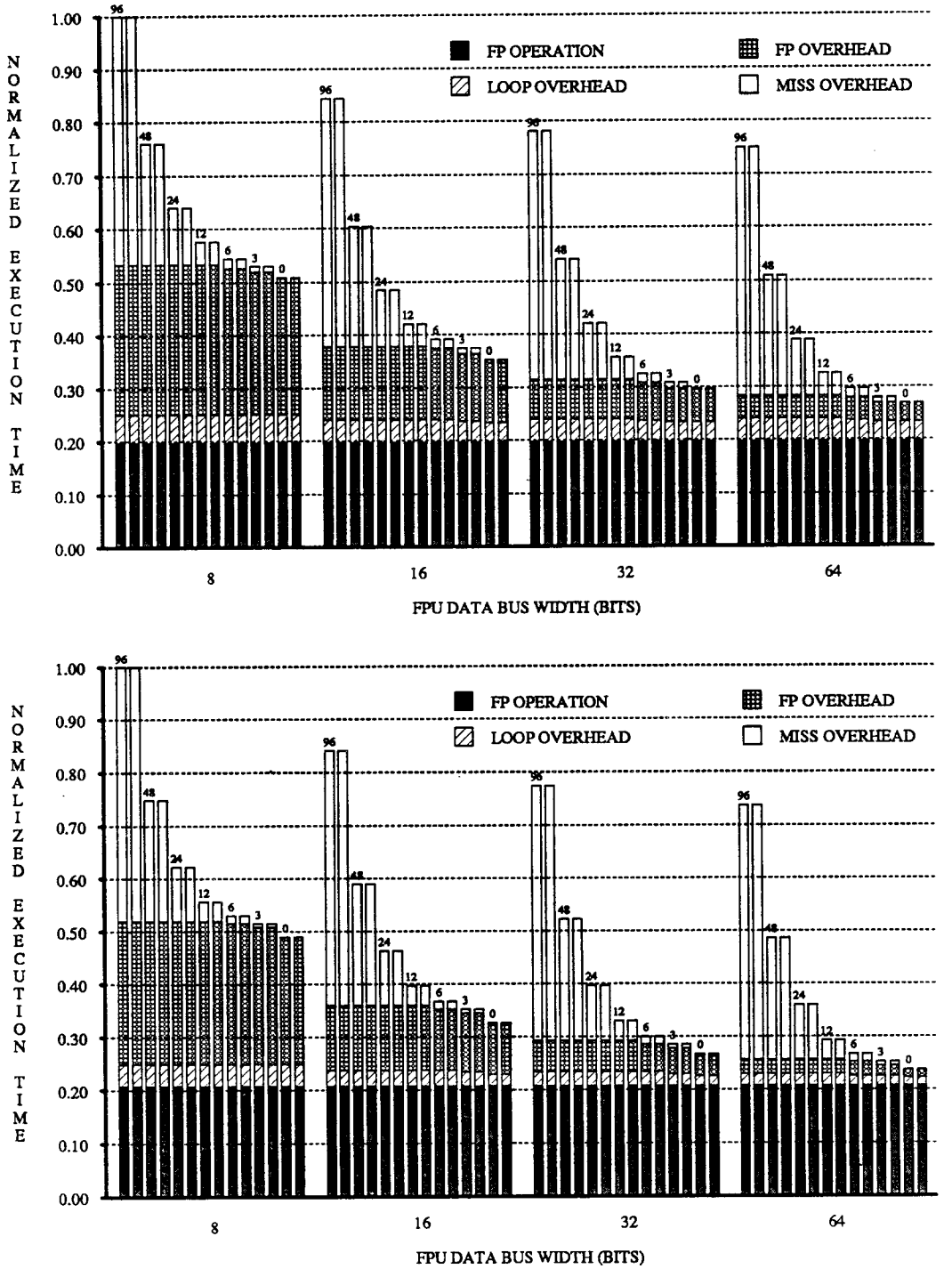


Figure B-7. SPUR System Performance for GE Programs.

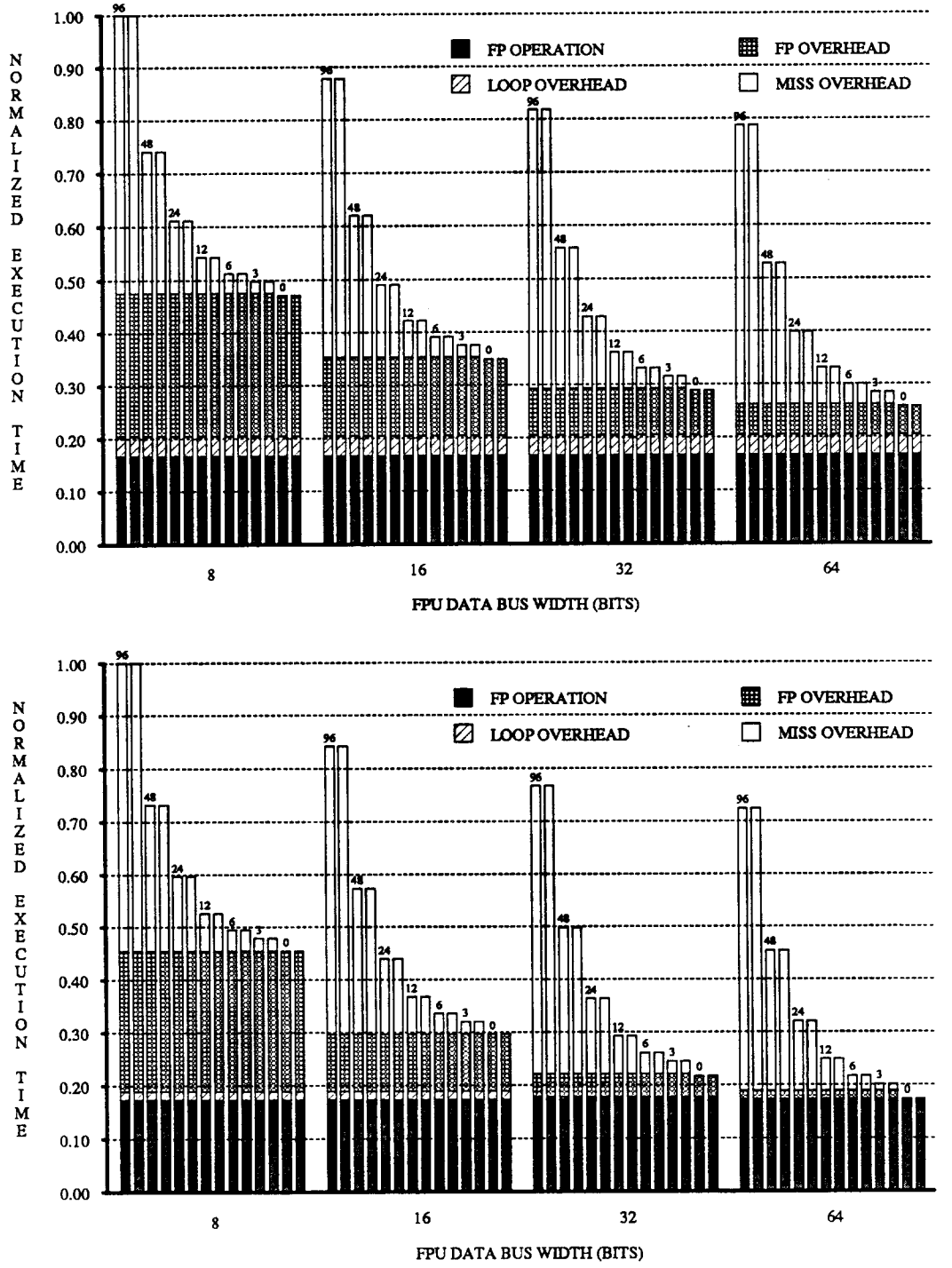


Figure B-8. SPUR System Performance for DP Programs.

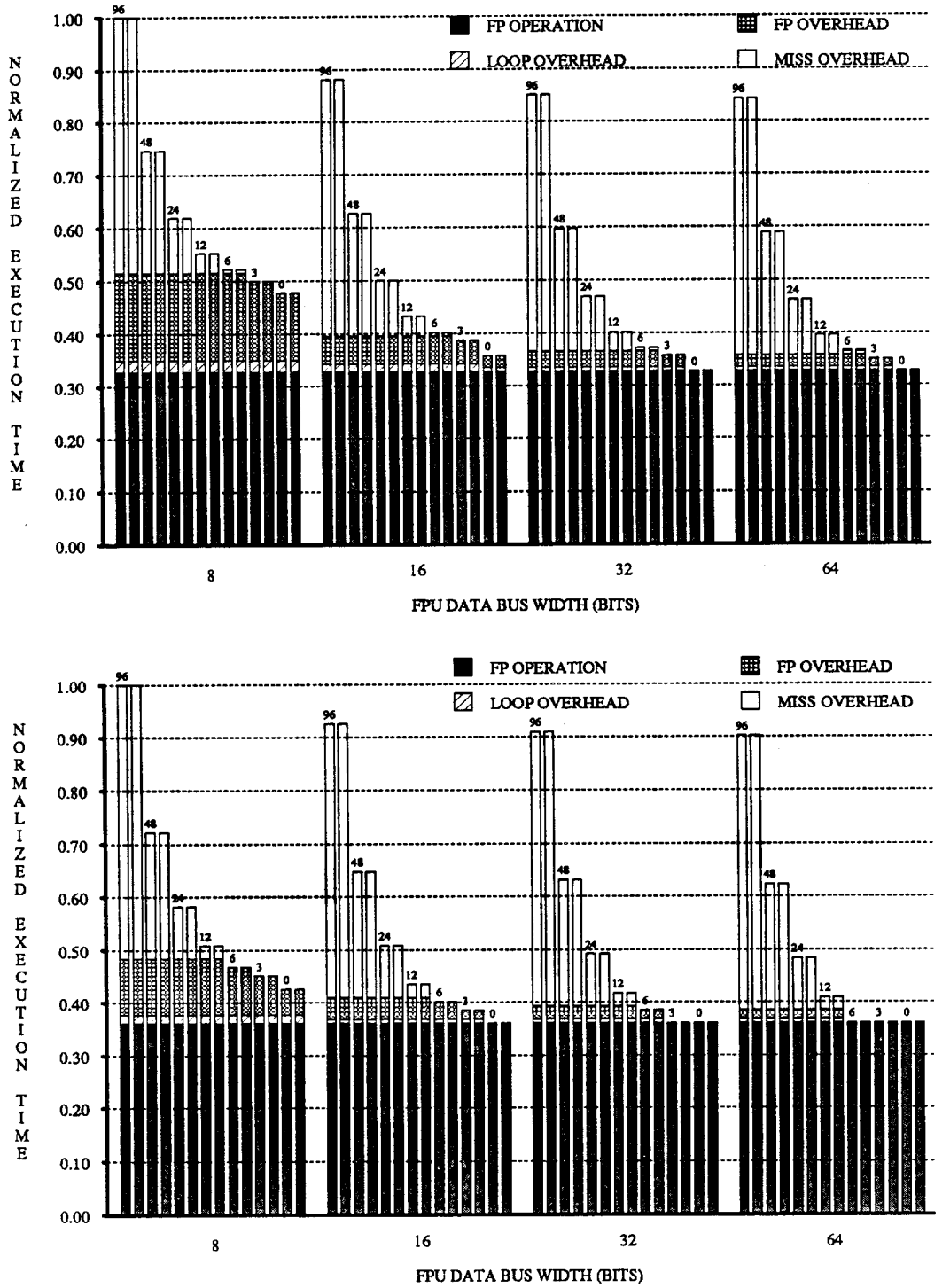


Figure B-9. SPUR System Performance for PE Programs.

B.5. Floating-point Instruction Times for Commercial Coprocessors

Included below are the instruction times for the Intel i8087 and i80287 Numeric Data Processors and the Motorola MC68881 Floating-point Coprocessor. These have been taken from [Inte85a, Inte85b, Moto85] and are expressed as *clock ticks* — the number of clock cycles to complete an operation. In some cases, the data books specify *clock counts*, which are multiple-clock-tick events. To determine the execution time, simply multiply the number of clock ticks by the assumed clock cycle period.

B.6. Intel i8087 or i80287 Floating-point Instruction Times

Table B-21. Intel i8087 and i82087 Instruction Times

Floating-point Operation	Clock Cycles			
	Reg-Reg	Mem-Reg-Sgl	Mem-Reg-Dbl	Mem-Reg-Ext
FLD	34-44	76-112	80-120	106-130
FST	30-44	168-180	192-208	-
FADD/FSUB	140-200	180-240	190-250	-
FMUL	180-290	220-250	224-336	-
FDIV	386-406	430-450	440-460	-
FSIN/FCOS	-	-	-	-
FSINCOS	-	-	-	-
FPTAN	60-1080	-	-	-
FSQT	360-372	-	-	-
FASIN	-	-	-	-
FACOS	-	-	-	-
FPATAN	500-1600	-	-	-

Assumptions for Intel FPU's:

1. 8 MHz, 10 MHz, and 12 MHz parts are available from Intel. Faster equivalent parts are available from second source vendors.
2. The above numbers do not account for effective address calculation time for memory-held operands, which varies between five and 12 clock-ticks. On average, eight is a reasonable estimate.
3. The above numbers do not account for bus cycles needed to fetch memory-held operands. Each operation which uses operands from memory consumes between two and six bus cycles. Each bus cycle costs four clock-ticks. Thus, each memory reference could consume between four and 24 clock-ticks. These times and amounts are given in the data book for each instruction and vary according to the alignment of words in memory. Odd-address aligned data requires typically two more bus cycles than even-address aligned data, or eight clock ticks.
4. The compiler generally inserts a "CPU WAIT" instruction in front of **every** FPU instruction. This costs (3 + 5n) clock-ticks, where n = number of times the cpu examines the TEST line before the i8087 lowers BUSY. The above numbers DO account for the *average* WAIT encountered (5/2 clock-ticks according to the Intel databook).
5. The above times assume all FPU instruction fetching is overlapped with CPU operations, so that no fetch-cost is accounted for in the above times.

6. The above numbers account for the average amount of time used for local bus arbitration, execution of the ESC instruction, and average WAIT (as mentioned before).
7. The low end of the clock-cycle range for the i8087 can be as much as 80 clock cycles slower than the i80287 for FMUL reg-reg and double precision mem-reg operations.

B.7. Motorola MC68881 Floating-point Instruction Times

Table B-22. Motorola MC68881 Instruction Times

Floating-point Operation	Clock Cycles			
	Reg-Reg	Mem-Reg-Sgl	Mem-Reg-Dbl	Mem-Reg-Ext
FMOVE(in)	33	52	58	56
FADD/FSUB	51	72	78	76
FSGLDIV	69	90	96	94
FSGLMUL	59	80	86	84
FDIV	103	124	130	128
FMUL	71	92	98	96
FSIN/FCOS	581	410	416	414
FSINCOS	451	470	476	474
FTAN	473	492	498	496
FASIN	581	600	606	604
FACOS	625	644	650	648
FATAN	403	422	428	426

Assumptions for Motorola FPU:

1. 12 MHz, 16 MHz, 20 MHz, and 25 MHz parts are available.
2. The above numbers do not account for effective address calculation time for memory-held operands, which varies between zero and 24 clock-ticks.
3. The above numbers do not account for bus cycles needed to fetch memory-held operands. Each operation which uses operands from memory consumes between two and four bus cycles. Each bus cycle costs a minimum of four clock-ticks. Thus, each memory reference could consume between eight and 16 clock-ticks. These times and amounts are given in the data book for each instruction and vary according to the alignment of words in memory. Odd-address aligned data typically requires more bus cycles than even-address aligned data.
4. The above times assume all FPU instruction fetching is overlapped with CPU operations, so that no fetch-cost is accounted for in the above times.
5. The above numbers account for the average amount of time used for local bus arbitration and execution of the read response register operation.

C Path Optimization Simulation Results

This appendix contains the simulation results for various scan-based algorithms described in Chapter 5. Ten maps were solved for each of six TCM sizes and all algorithms. Each table entry lists the algorithm number, the size of the TCM array, the lowest, highest, mean, median, and mode number of cell checks, standard deviation, and coefficient of variation for the number of cell checks for the 10 maps. The manual pages for both the scan-based simulator *scan* and Dijkstra's Algorithm *dij* are included in Section C.2 and Section C.3.

C.1. Scan-based Algorithm Simulation Results

Table C-1 and Table C-2 report results for 4-way tests with the goal point at [1,3] and [C,C] respectively. Table C-3 and Table C-4 report results for 8-way tests with the goal point at [1,3] and [C,C] respectively.

C.1.1. Four-way Cell Check with Goal at TCM[1,3]

Table C-1. Cell Checks: 4-Way Test, Goal at [1,3], Grouped by Algorithm, 10 Maps

Alg	Size	Low	High	Mean	Median	Mode	Std Dev	Coeff Var
2	008	144	252	172.8	144	144	40.9	23.65
2	016	980	1568	1391.6	1372	1568	215.7	15.50
2	064	61504	99944	84952.4	84568	84568	11944.6	14.06
2	128	428652	857304	666792.0	682668	698544	114483.4	17.17
2	256	4903216	5741924	5251602.5	5290312	5290312	234006.3	4.46
2	512	34853400	46297800	42318272.0	41616000	41355900	3209186.5	7.58
3	008	72	132	114.0	132	132	29.0	25.42
3	016	756	1134	831.6	756	756	159.4	19.2
3	032	1800	5280	3540.0	3540	3540	820.2	23.17
3	064	15252	53072	25841.6	22816	15252	12954.9	50.13
3	128	63252	346752	214452.0	220752	346752	103732.4	48.37
3	256	900176	1928368	1542796.0	1671320	1671320	277643.1	18.00
3	512	10903800	17133960	13395864.0	12980520	11942160	2030042.3	15.15
4	008	144	252	172.8	144	144	40.9	23.65
4	016	1176	1568	1411.2	1372	1568	180.1	12.76
4	032	9000	15300	11070.0	10800	9000	2036.9	18.40
4	064	69192	103788	86105.6	84568	84568	12316.8	14.30
4	128	603288	825552	684255.6	666792	666792	69986.6	10.23
4	256	4774184	5677408	5116119.0	5032248	5032248	296510.0	5.80
4	512	35633700	43176600	40575604.0	40835700	41095800	2300356.0	5.67
5	008	126	186	132.0	126	126	19.0	14.37
5	016	742	2380	1415.4	1106	1106	569.4	40.23
5	032	5250	24390	16212.0	17430	20910	6408.9	39.53
5	064	121086	162688	138105.0	136214	121086	17216.3	12.47
5	128	1039626	1590876	1263276.0	1260126	1260126	194576.0	15.40
5	256	8225790	11952986	10577778.0	10539222	11952986	1187343.4	11.22
5	512	69570632	91376192	81771360.0	79954232	91376192	6945050.5	8.49
6	008	144	216	176.4	144	144	35.8	20.29
6	016	980	2156	1568.0	1568	1764	357.9	22.82
6	032	8100	18900	12690.0	10800	10800	3781.7	29.80
6	064	73036	115320	88027.6	84568	92256	13248.0	15.05
6	128	460404	714420	625514.4	635040	635040	73022.0	11.67
6	256	4129024	5290312	4787087.0	4774184	5032248	332882.2	6.95
6	512	30951900	42136200	35451632.0	34593300	39015000	3643540.3	10.28
7	008	108	216	151.2	144	108	44.3	29.27
7	016	1176	1568	1332.8	1372	1372	124.0	9.30
7	032	9000	16200	11250.0	10800	9000	2333.4	20.74
7	064	65348	111476	87643.2	88412	88412	14920.8	17.02
7	128	587412	746172	662029.2	635040	746172	58476.0	8.83
7	256	4838700	5483860	5161280.0	5096764	5096764	194733.1	3.77
7	512	36414000	44477100	41459936.0	41616000	41616000	2403388.8	5.80
8	008	108	216	154.8	144	108	41.7	26.97
8	016	784	2156	1489.6	1372	1372	394.2	26.46
8	032	8100	19800	12510.0	11700	16200	3944.7	31.53
8	064	65348	115320	86490.0	84568	92256	14970.2	17.31
8	128	460404	682668	614401.2	619164	619164	65693.4	10.69
8	256	4129024	5096764	4729023.0	4774184	4774184	294945.5	6.24
8	512	31212000	42136200	35087492.0	33032700	32252400	3414353.0	9.73
9	008	140	245	196.0	175	175	37.6	19.20
9	016	1755	3315	2301.0	1950	1950	534.4	23.23
9	032	14384	25172	20766.9	20677	20677	3753.6	18.08
9	064	126819	249795	205216.2	219051	219051	42686.3	20.80
9	128	1111250	2079625	1647825.0	1730375	1730375	338567.0	20.55
9	256	9290160	14580390	12419138.0	11677215	11677215	1979678.0	15.94
9	512	102479008	137072176	120894016.0	123286928	113663264	12661642.0	10.47

C.1.2. Four-way Cell Check with Goal at TCM[C,C]

Table C-2. Cell Checks: 4-Way Test, Goal at [C,C], Grouped by Algorithm, 10 Maps

Alg	Size	Low	High	Mean	Median	Mode	Std Dev	Coeff Var
2	008	144	252	172.8	144	144	40.9	23.65
2	016	980	1568	1391.6	1568	1568	215.7	15.50
2	032	9000	12600	10980.0	10800	10800	1106.3	10.08
2	064	61504	99944	84952.4	88412	84568	11944.6	14.06
2	128	428652	857304	666792.0	698544	698544	114483.4	17.17
2	256	4903216	5741924	5251602.5	5290312	5290312	234006.3	4.46
2	512	34853400	46297800	42318272.0	42656400	41355900	3209149.3	7.58
3	008	72	132	114.0	132	132	29.0	25.42
3	016	756	1134	831.6	756	756	25401.6	159.4
3	032	1800	5280	3540.0	3540	3540	820.2	23.17
3	064	15252	53072	25841.6	22816	15252	12954.9	50.13
3	128	63252	346752	214452.0	220752	63252	103732.3	48.37
3	256	900176	1928368	1542796.0	1671320	1671320	277643.1	18.00
3	512	10903800	17133960	13395864.0	12980520	11942160	2030046.0	15.15
4	008	144	252	172.8	144	144	40.9	23.65
4	016	1176	1568	1411.2	1568	1568	180.1	12.76
4	032	9000	15300	11070.0	10800	9000	2036.9	18.40
4	064	69192	103788	86105.6	84568	84568	12316.8	14.30
4	128	603288	825552	684255.6	666792	666792	69987.0	10.23
4	256	4774184	5677408	5116119.0	5032248	5032248	296510.0	5.80
4	512	35633700	43176600	40575600.0	41095800	40575600	2300397.5	5.67
5	008	126	186	132.0	126	126	19.0	14.37
5	016	742	2380	1415.4	1470	1106	569.4	40.23
5	032	5250	24390	16212.0	19170	17430	6408.9	39.53
5	064	121086	162688	138105.0	136214	121086	17216.3	12.47
5	128	1039626	1590876	1263276.0	1260126	1039626	194576.0	15.40
5	256	8225790	11952986	10577778.0	10667746	11952986	1187337.0	11.22
5	512	69570632	91376192	81771368.0	81511768	69570632	6944995.5	8.49
6	008	144	216	176.4	180	144	35.8	20.29
6	016	980	2156	1568.0	1568	1372	357.9	22.82
6	032	8100	18900	12690.0	12600	9000	3781.7	29.80
6	064	73036	115320	88027.6	92256	92256	13248.0	15.05
6	128	460404	714420	625514.4	635040	619164	73022.0	11.67
6	256	4129024	5290312	4787087.0	4838700	4129024	332879.3	6.95
6	512	30951900	42136200	35451628.0	35893800	30951900	3643579.8	10.28
7	008	108	216	151.2	144	108	44.3	29.27
7	016	1176	1568	1332.8	1372	1372	124.0	9.30
7	032	9000	16200	11250.0	10800	9000	2333.4	20.74
7	064	65348	111476	87643.2	88412	88412	14920.8	17.02
7	128	587412	746172	662029.2	666792	619164	58476.0	8.83
7	256	4838700	5483860	5161280.0	5161280	5096764	194733.1	3.77
7	512	36414000	44477100	41459936.0	42136200	41616000	2403388.8	5.80
8	008	108	216	154.8	144	108	41.7	26.97
8	016	784	2156	1489.6	1372	1372	394.2	26.46
8	032	8100	19800	12510.0	12600	8100	3944.7	31.53
8	064	65348	115320	86490.0	92256	92256	14970.2	17.31
8	128	460404	682668	614401.2	619164	619164	65693.4	10.69
8	256	4129024	5096764	4729023.0	4774184	4774184	294945.5	6.24
8	512	31212000	42136200	35087488.0	35893800	32252400	3414398.3	9.73
9	008	140	245	196.0	175	175	37.6	19.20
9	016	1755	3315	2301.0	2145	1950	534.4	23.23
9	032	14384	25172	20766.9	21576	20677	3753.6	18.07
9	064	126819	249795	205216.2	219051	219051	42686.3	20.80
9	128	1111250	2079625	1647825.0	1730375	1730375	338567.7	20.55
9	256	9290160	14580390	12419138.0	13999755	9290160	1979684.0	15.94
9	512	102479008	137072176	120894016.0	124587424	102479008	12661642.0	10.47

C.1.3. Eight-way Cell Check with Goal at TCM[1,3]

Table C-3. Cell Checks: 8-Way Test, Goal at [1,3], Grouped by Algorithm, 10 Maps

Alg	Size	Low	High	Mean	Median	Mode	Std Dev	Coeff Var
2	008	144	252	162.0	144	144	35.0	21.60
2	016	1176	2156	1666.0	1568	1568	295.8	17.76
2	032	9900	15300	13500.0	13500	12600	1643.2	12.17
2	064	96100	130696	107247.6	103788	103788	10480.3	9.77
2	128	777924	936684	874767.6	873180	857304	45182.9	5.17
2	256	6645148	8064500	7148373.0	7032244	7354824	400946.1	5.61
2	512	50459400	60343200	55245236.0	55401300	56181600	2656009.8	4.81
3	008	192	372	240.0	192	192	62.0	25.82
3	016	1848	3668	2685.2	2576	2576	486.9	18.13
3	032	17460	29640	24768.0	24420	24420	3460.6	13.97
3	064	181660	257300	203595.6	189224	189224	24038.7	11.81
3	128	1480752	1795752	1647702.0	1638252	1638252	86650.3	5.26
3	256	11953240	15808960	13906805.0	13881100	14266672	1002516.9	7.21
3	512	99683584	118893240	107678952.0	107990464	107990464	5451835.5	5.06
4	008	108	252	158.4	144	144	38.7	24.43
4	016	1176	2156	1803.2	1764	1960	303.6	16.84
4	032	13500	18000	15570.0	15300	14400	1532.6	9.84
4	064	111476	157604	131080.4	126852	126852	14882.2	11.35
4	128	952560	1190700	1085918.4	1079568	1063692	80327.0	7.40
4	256	7677404	10258044	8535466.0	8258048	9225788	763400.8	8.94
4	512	65545200	73348200	69654784.0	70487104	70487104	2315997.0	3.32
5	008	126	186	141.0	126	126	21.2	15.04
5	016	1470	2380	1925.0	1834	1834	300.3	15.60
5	032	13080	21780	18300.0	18300	20040	2751.2	15.03
5	064	132432	177816	152476.6	147560	162688	15649.5	10.26
5	128	1134126	1291626	1231776.0	1244376	1244376	43165.9	3.50
5	256	9189720	10474960	9748800.0	9703816	9768078	358491.9	3.68
5	512	72166528	81771360	77072784.0	76319968	79954232	3376726.3	4.38
6	008	108	216	154.8	144	144	34.1	22.06
6	016	1372	2352	1822.8	1960	1960	292.9	16.07
6	032	12600	20700	16920.0	16200	19800	2642.7	15.62
6	064	126852	169136	147994.0	142228	126852	16632.7	11.24
6	128	984312	1238328	1117670.4	1111320	1111320	81022.1	7.25
6	256	8387080	10387076	9122562.0	8903208	8838692	565058.0	6.19
6	512	67105800	73868400	70617144.0	70747200	71007296	2152824.0	3.05
7	008	144	216	165.6	144	144	34.8	21.00
7	016	1568	2352	1803.2	1568	1568	303.6	16.84
7	032	9900	17100	13500.0	13500	9900	2545.6	18.86
7	064	88412	134540	103403.6	96100	115320	14320.0	13.85
7	128	650916	920808	774748.8	746172	793800	86244.9	11.13
7	256	5161280	6903212	5858053.0	5677408	5225796	609628.9	10.41
7	512	39535200	46297800	43592760.0	44217000	45777600	2785687.0	6.39
8	008	108	252	176.4	180	180	43.1	24.43
8	016	1568	2548	1940.4	1764	1568	374.7	19.31
8	032	11700	17100	14580.0	14400	15300	1839.6	12.62
8	064	103788	138384	118779.6	119164	123008	9319.4	7.85
8	128	746172	1047816	920808.0	904932	968436	86957.0	9.44
8	256	6387084	8580628	7174179.0	6709664	6709664	722141.3	10.07
8	512	47338200	62944200	56883872.0	57742200	54360900	4289747.0	7.54
9	008	105	210	143.5	140	140	38.5	26.84
9	016	975	2145	1540.5	1755	1755	394.8	25.63
9	032	8990	18879	13305.2	12586	15283	3249.7	24.42
9	064	76860	119133	99918.0	103761	107604	15157.0	15.17
9	128	682625	1031875	860425.0	857250	952500	120157.1	13.96
9	256	5935380	8709525	7225680.0	7032135	6580530	832888.3	11.53
9	512	43956732	64764652	56311436.0	56961680	58262176	5688224.5	10.10

C.1.4. Eight-way Cell Check with Goal at TCM[C,C]

Table C-3. Cell Checks: 8-Way Test, Goal at [C,C], Grouped by Algorithm, 10 Maps

Alg	Size	Low	High	Mean	Median	Mode	Std Dev	Coeff Var
2	008	144	216	180.0	180	180	17.0	9.43
2	016	1372	2156	1705.2	1568	1568	262.1	15.37
2	032	10800	18900	14400.0	15300	11700	2510.0	17.43
2	064	99944	142228	116088.8	115320	111476	13414.3	11.56
2	128	793800	920808	881118.0	889056	889056	41162.6	4.67
2	256	6258052	7161276	6670954.5	6774180	6903212	290440.8	4.35
2	512	10000000	56181600	52410148.0	52540200	51239700	1668946.3	3.18
3	008	252	312	300.0	312	312	25.3	8.43
3	016	1848	2940	2430.4	2576	2576	306.9	12.63
3	032	17460	20940	19548.0	19200	19200	1372.5	7.02
3	064	128712	174096	149134.8	143840	143840	16359.5	10.97
3	128	882252	1197252	1052352.0	1197252	1102752	105209.9	10.00
3	256	7197852	8740140	7801915.0	7840472	7583424	477248.3	6.12
3	512	10000000	61264260	57422328.0	57110820	57110820	1867002.9	3.25
4	008	144	216	187.2	180	180	22.8	12.16
4	016	1176	2156	1587.6	1568	1372	284.0	17.89
4	032	9900	16200	12960.0	13500	12600	1654.1	12.76
4	064	96100	126852	107247.6	96100	111476	10480.3	9.77
4	128	714420	825552	766810.8	762048	762048	35930.7	4.69
4	256	5032248	6322568	5735472.5	5806440	5806440	351330.8	6.13
4	512	10000000	46037700	44581144.0	44737200	43696800	902585.1	2.02
5	008	156	186	171.0	186	156	15.8	9.25
5	016	1470	1834	1597.4	1652	1652	122.8	7.69
5	032	10470	14820	12993.0	13080	13080	1192.2	9.18
5	064	94612	109740	101797.8	109740	105958	5182.6	5.09
5	128	677376	882126	748251.0	756126	756126	58579.1	7.83
5	256	5076952	6169406	5507507.0	5462524	5076952	342801.5	6.22
5	512	10000000	44390400	42780944.0	43611632	44390400	1589899.1	3.72
6	008	144	216	180.0	180	180	17.0	9.43
6	016	1372	1764	1470.0	1372	1372	138.6	9.43
6	032	10800	15300	12960.0	13500	11700	1481.9	11.43
6	064	80724	103788	94178.0	92256	92256	7526.1	7.99
6	128	603288	777924	689018.4	682668	666792	48038.3	6.97
6	256	4774184	5935472	5154828.5	5032248	4838700	382831.3	7.43
6	512	10000000	44737200	38572832.0	38494800	36934200	2380815.8	6.17
7	008	144	216	194.4	216	216	25.2	12.95
7	016	1176	2352	1705.2	1764	1568	307.1	18.01
7	032	11700	15300	13680.0	13500	13500	1394.3	10.19
7	064	88412	103788	99944.0	96100	103788	4794.2	4.80
7	128	587412	809676	720770.4	714420	714420	66603.6	9.24
7	256	5032248	5999988	5522569.5	5612892	5741924	292033.3	5.29
7	512	10000000	46297800	43410692.0	43436700	43176600	1665253.6	3.84
8	008	144	216	190.8	180	180	24.3	12.73
8	016	1176	1960	1607.2	1568	1568	202.4	12.60
8	032	11700	15300	12870.0	13500	11700	1203.7	9.35
8	064	80724	115320	93793.6	92256	92256	9450.6	10.08
8	128	555660	666792	606463.2	635040	555660	51742.5	8.53
8	256	3870960	4709668	4232249.5	4193540	4193540	247451.8	5.85
8	512	10000000	38754900	33891032.0	34073100	28871100	2580830.0	7.62
9	008	105	175	140.0	140	140	28.6	20.41
9	016	975	1560	1287.0	1365	1170	209.6	16.29
9	032	8990	9889	9529.4	9889	9889	464.2	4.87
9	064	61488	92232	79550.1	80703	80703	8884.3	11.17
9	128	460375	587375	546100.0	571500	539750	41800.9	7.65
9	256	3354780	4580565	3922511.5	3999930	3999930	398631.1	10.16
9	512	10000000	34593168	32174246.0	33032572	34333068	2230878.3	6.93

C.2. Scan-based Path Optimization Simulator Manual Page

SCAN(LOCAL) USER COMMANDS SCAN(LOCAL)

NAME

scan – simulator for scan-based path optimization algorithms

SYNOPSIS

scan [options]

DESCRIPTION

Scan is a C-language simulator designed to explore various algorithms and techniques used in a dynamic programming implementation of path planning optimization. The program allows a large number of user specified parameters for input variables, diagnostic functions, and output information. *Scan* can either generate its own DCM data, as specified by a command line option, or read DCM data sets. It reads files as specified and either writes to standard output or files named in the command line. The following command-line options are understood:

-a *N* Select algorithm *N* for the simulation run. See header file *scan.h* for a complete list.

-b *N* Specify the beginning map number *N*. This allows a simulation which terminates after running several maps to be re-run at a later time generating the same output as would have been created had the program continued initially.

-c A debugging feature for the Algorithm No. 9, showing the addresses considered during the *scan*'ing processing.

-d Enable debug. As a result, many parameters and a lot of information about the program as it runs will be printed to standard out.

-e *N* Specify the ending map number *N*(see **-b** above).

-g *outTcm*
Write TCM array in gremlin format to file *outTcm*.

-i *inFile*
Read in DCM data sets from file *inFile*, data rather than create it using the random number generator. If the user selects this option, the concatenated DCM's (in the order NE, NE, N, NE, E, SE, S, SW, W) must be in one single file, name following the option switch.

-k
Change some elements in the TCM after the solution has been found. This allows the user to determine if the algorithm operates correctly in finding the TCM solution following a single pass with no exchanges.

-l *N* *N* is the number of no-change sweeps (a *macro-sweep*) that occur before the algorithm terminates because of convergence.

-m *N* *N* is the maximum DCM value, within the range [1 .. 255].

-o *tcmFile*
Write the TCM array into *tcmFile*.

-n *N* *N* specify a template which will be used to determine which of the eight neighbor cells will be considered in determining the minimum cost cell for the center. The template is represented by three numbers. The first number is the decimal equivalent of the binary number derived by writing in order 1's for each of the cells which will be considered for the minimization function. For example, if the NW, N, and NE cells are all to be considered, the decimal equivalent number of 7 would be the first digit of the template value. If the W and the E cells are to be considered, the next decimal number in the template would be 5, and if the SW, S, and SE cells are to be considered, the last number in the template would again be 7.

-p *dcmFile*
Write the DCM arrays into *dcmFile*.

-q *updateMap*
For algorithm debugging purposes. A map representing the cells of the TCM will be written to *updateMap* each sweep. A 1 in *updateMap* represents a cell where an update was made this sweep; a 0 represents a cell in which there was no update.

-r *N* *N* is the seed supplied to the random number generator routine.

-s *inGoalPoints*
Open and read the file *inGoalPoints*, which should be pairs of [row][column] coordinates that are the goal points to use in solving the TCM. Besides reading the goal coordinates, the DCM's will also be read.

-v
Print a map of the same dimensionality as the TCM indicating the order in which cells have been checked. For most scan-based algorithms, this has no particular utility. For some algorithms, this is useful in debugging.

-w *swUpdateFile*
Write the number of updates performed during a given sweep of the algorithm to file *swUpdateFile*. The numbers in the file which are written are: (1) sweep number, followed by (2) the number of exchanges during that sweep.

-x *N* *N* is the goal point row number, in the range [0 .. N-1].

-y *N* *N* is the goal point column number, in the range [0 .. N-1].

-z *N* *N* is the size of the TCM. The smallest map is a 4-by-4 array of elements, and the largest map is an array of size 512-by-512 elements. Larger maps take a considerable amount of time to solve.

-h
The *help* option. Lists the command line options and their parameters.

FILES

/usr/name/scan/scan.h	header file containing definitions
/usr/name/scan/*.c	source files (approx. 3150 lines)

SEE ALSO

gremlin(LOCAL), ditroff(1)

AUTHOR

Paul M. Hansen

C.3. Dijkstra's Algorithm Path Optimization Simulator Manual Page

DIJ(LOCAL) USER COMMANDS DIJ(LOCAL)

NAME

dij – simulator for Dijkstra's Shortest Path Algorithm

SYNOPSIS

dij [options]

DESCRIPTION

Dij is a C-language implementation of Dijkstra's Shortest Path Algorithm applied to path planning optimization. The program allows a large number of user specified parameters for input variables, diagnostic functions, and output information. *Dij* can either generate its own DCM data, as specified by a command line option, or read DCM data sets. It reads files as specified and either writes to standard output or files named in the command line. The following command-line options are understood:

-b *N* Specify the beginning map number *N*. This allows a simulation which terminates after running several maps to be re-run at a later time generating the same output as would have been created had the program continued initially.

-c *inTcmFile*
Read in TCM map from file *inTcmFile*. Useful when planning a path through a previously solved TCM.

-d Enable debug. As a result, many parameters and a lot of information about the program as it runs will be printed on standard out.

-e *N* Specify the ending map number *N*(see **-b** above).

-f Simulate using a four-near-neighbor test. Otherwise, do 8-way tests.

-g *pathFile*
Write a gremlin-format file *pathFile* that plots the path from start to goal point.

-i *inFile*
Read in DCM data sets from file *inFile*, data rather than create it using the random number generator. If the user selects this option, the concatenated DCM's (in the order NE, NE, N, NE, E, SE, S, SW, W) must be in one single file, name following the option switch.

-m *N* *N* is the maximum DCM value, within the range [1 .. 255].

-n Find and print the maximum value of TCM array.

-o *tcmFile*
Write the TCM array into *tcmFile*.

-p Print the path on standard out using line printer format.

-q *dcmFile*
Write the DCM arrays into *dcmFile*.

-r *N* *N* is the seed supplied to the random number generator routine.

-s *inStartPoints*
Open and read the files *inStartPoints* and *inGoalPoints*, which should be pairs of [row][column] coordinates that are the start and goal points to use in solving the TCM. Besides reading the start/goal coordinates, the DCM's will also be read.

-t
Print the TCM on standard out.

-u *N*
If *N* is 1, start an interactive session; otherwise, this is a batch simulation run.

-x *N* *N* is the goal point row number, in the range [0 .. N-1].

-y *N* *N* is the goal point column number, in the range [0 .. N-1].

-z *N* *N* is the size of the TCM. The smallest map is a 4-by-4 array of elements, and the largest map is an array of size 512-by-512 elements. Larger maps take a considerable amount of time to solve.

-h
The *help* option. Lists the command line options and their parameters.

FILES

/usr/name/dij/dij.h	header file containing definitions
/usr/name/dij/*.c	source files (approx. 2700 lines)

SEE ALSO
gremlin(LOCAL), ditroff(1)

AUTHOR
Paul M. Hansen

<This page is intentionally blank.>

D Digital Signal Processing Chips with Floating-point Arithmetic

This appendix contains a list of several manufactures who already or soon will offer DSP chips that support floating-point arithmetic. The second section illustrates a block diagram of the AT&T DSP32 architecture, one of the most popular commercial DSP chips supporting floating-point operations.

D.1. Commercial DSP Chips with Floating-point Arithmetic

This table lists several parameters for each of six DSP chips that are now or will soon be offered commercially [Wils88]. An entry marked "na" is used for information that was not available at the time of this writing, does not necessarily imply that the given device does not support or include that parameter.

Table D-1. Commercial Digital Signal Processing Chips Supporting Floating-point Arithmetic

Parameter	Manufacturer					
	AT&T	AT&T	Fujitsu	Moto	NEC	TI
Name	DSP32	DSP32C	MB86232	96002	μPD77230	320C30
Year avail	1/85	89	12/88	6/89	86	87
Technology	NMOS	CMOS	CMOS	HCMOS	CMOS	CMOS
Line width	1.5 μ	.75 μ	1.5 μ	1.2 μ	1.5 μ	1.0 μ
Cycle time	250 nsec	80 nsec	75 nsec	75 nsec	150 nsec	60 nsec
FP Data	32-bit	32-bit	24-bit	32-bit	32-bit	32-bit
Addr bits	16	24	24	24	32	24
1024 pt. FFT	6.7 msec	2.2 msec	2.2 msec	2.0 msec	4.7 msec	1.67 msec
Peak MIPS	4.0	12.5	12.5	13.3	6.7	16.7
Peak MFLOPS	8	25	na	40	na	33
RAM	2@512x32	2@1Kx32	na	na	2@512x32	1@2Kx32
ROM	1@512x32	1@4Kx32	na	na	1@1Kx32, 1@2Kx32	1@4Kx32

This table lists several commercial digital signal processing chips that provide floating-point arithmetic. The processing rates reflect the current state of the art in VLSI and specialized architectures.

D.2. The AT&T DSP32 Digital Signal Processing Chip Architecture

AT&T was the first commercial manufacturer to offer a 32-bit floating-point DSP chip. Used internally at AT&T since 1985, it is still one of the few devices in volume production. The DSP32 has one parallel port, two serial ports, a 64-Kbyte program/address space, byte addressability, concurrent DMA, and the memory is not partitioned — it can be used for either data or instructions [Brod85]. The next generation DSP32C will have an 80-nanosecond cycle time, increased addressing capability (24-bit addresses), be able handle interrupts more efficiently, and will allow flexibility in interfacing to different speed memories.

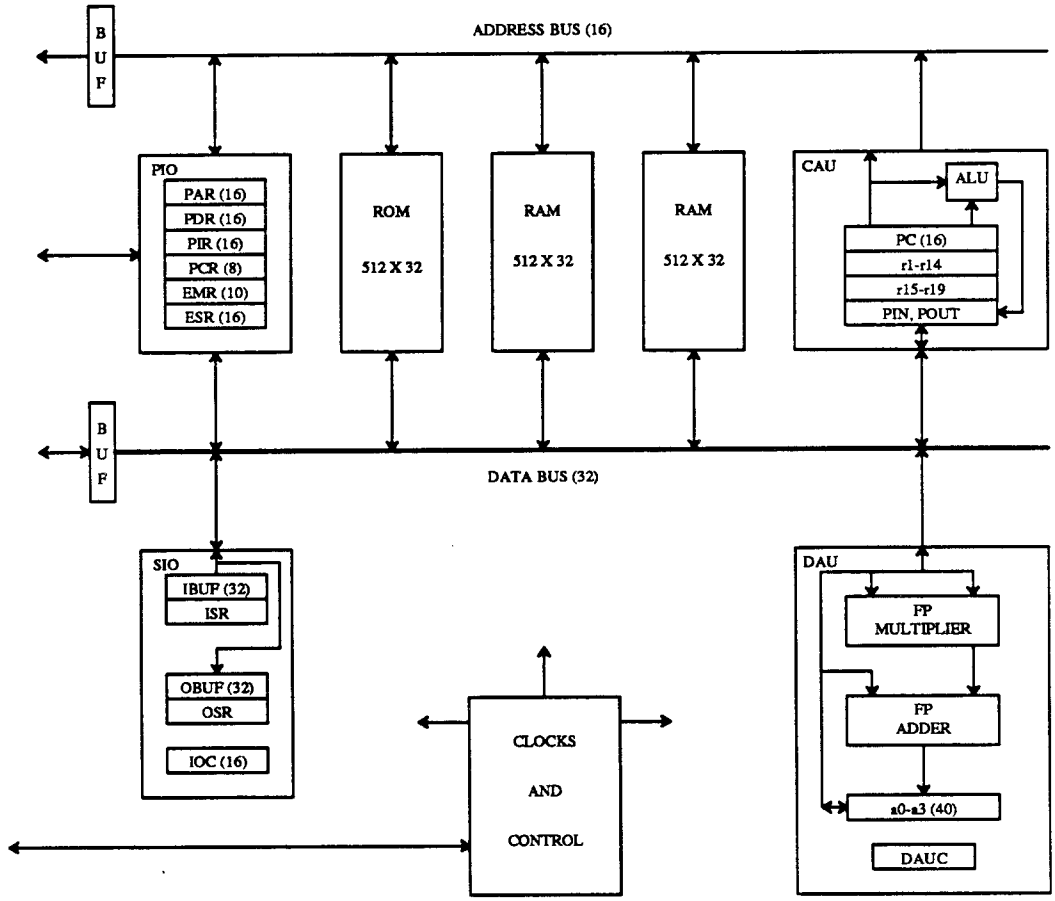


Figure D-1. Block Diagram of the AT&T DSP32 Architecture.

This figure shows a simplified block diagram of the AT&T DPS32 digital signal processing chip. The system allows concurrent operation between the integer instruction unit and the data arithmetic unit (DAU).

<This page is intentionally blank.>

References

[ASPL82] ASPLOS-I, Conference Proceedings, *Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1-3 1982.

[ASPL87] ASPLOS-II, Conference Proceedings, *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 5-8, 1987.

[Aho76] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1976.

[Amda67] G. M. Amdahl, Validity of the Single-processor Approach to Achieving Large Scale Computing Capabilities, *AFIPS Conference Proceedings*, Vol. 30, April 18-20, 1967, pp. 483-485, AFIPS Press, Reston, VA.

[Ande67] D. W. Anderson, F. J. Sparacio and F. M. Tomasulo, The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, *IBM Journal*, Vol. 11, January 1967, pp. 8-24.

[Ande87] D. V. Anderson, A. E. Koniges and A. A. Mirin, Multiprocessing Algorithms for the Cray-2, Institutional Research and Development, Report UCRL-53689-87, Lawrence Livermore National Laboratory, Livermore, CA, 1987.

[Andr88] W. Andrews, DSP Applications Ride the Wave of Floating-point Processing, *Computer Design*, Vol. 27, No. 16, September 1, 1988, pp. 50-61.

[Anon85] Anonymous, Database Sorter, *Datamation*, May 1, 1985, pp. 150.

[Arno83] R. D. Arnold, Automated Stereo Perception, STAN-CS-83-961, Department of Computer Science, Stanford, CA, March 1983.

[Asai86] S. Asai, Semiconductor Memory Trends, *Proceedings of the IEEE*, Vol. 74, No. 12, December 1986, pp. 1623-1635.

[Baas78] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, 1978.

[Bacc84] G. Baccarani, M. R. Wordeman and R. H. Dennard, Generalized Scaling Theory and Its Application to a 1/4 Micrometer MOSFET DESIGN, *IEEE Transactions on Electron Devices*, Vol. ED-31, No. 4, 1984, pp. 452-462.

[Bata82] J. Batali and *et al.*, The Scheme-81 Architecture — System and Chip, *Proceedings of the Conference on Advanced Research in VLSI*, 1982, pp. 69-77.

[Bell78] C. G. Bell, J. C. Mudge and J. E. McNamara, The PDP-8 and Other 12-Bit Computers in *Computer Engineering - A DEC View of Hardware Systems Design*, Bedford, Massachusetts, September 1978, pp. 175-208.

[Bell84] C. G. Bell, The Mini and Micro Industries, *Computer*, Vol. 17, No. 10, October 1984, pp. 14-30.

[Bell86] C. G. Bell, Personal Letter, May 10, 1986.

[Bell58] R. E. Bellman, On a Routing Problem, *Quarterly of Applied Mathematics*, No. 16, 1958, pp. 87-90.

[Bell65] R. Bellman and R. Kalaba, *Dynamic Programming and Modern Control Theory*, Academic Press, New York, 1965.

[Birn85] J. S. Birnbaum and W. S. Worley, Jr., Beyond RISC: High-Precision Architecture, *Hewlett-Packard Journal*, Vol. 36, No. 8, August 1985, pp. 4-10.

[Bizj86] C. Bizjak, Unpublished data from personal communication, Sun Microsystems, June 1986. Reports experience with the Sun Microsystems *bitblt* chip.

[Blaa64] G. A. Blaauw and F. P. Brooks, The Structure of SYSTEM/360, *IBM Systems Journal*, Vol. 3, No. 2, 1964, pp. 119-135.

[Bori88] G. Boriello, A New Interface Specification Methodology and its Application to Transducer Synthesis, Computer Science Division (EECS) Report No. UCB/CSD 88/430, University of California, Berkeley, CA, May 26, 1988. Ph.D. Dissertation.

[Borr85] G. Borriello, R. Katz, A. Bell and L. Conway, VLSI Design 'By The Numbers', *IEEE Spectrum*, Vol. 22, No. 2, February 1985.

[Borr87] G. Borriello, A. Cherenson, P. Danzig and M. Nelson, RISCs or CISCs for Prolog: A Case Study, *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 5-8, 1987.

[Bose85] P. Bose and E. S. Davidson, Design of Instruction Set Architectures for Support of High-level Languages, *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, June 1985, pp. 198-206.

[Bose88a] B. K. Bose, P. M. Hansen, C. Lee and D. A. Patterson, Fast Scientific Computation in CMOS VLSI Shared-Memory Multiprocessors, *1988 IEEE International Symposium on Circuits and Systems (ISCAS'88)*, Helsinki, Finland, June 1988, pp. 811-814.

[Bose88b] B. K. Bose, VLSI Design Techniques for Floating-Point Computation, Computer Science Division (EECS) Report No. UCB/CSD TBD, University of California, Berkeley, CA, December 1988. Ph.D. Dissertation.

[Bose88c] B. K. Bose, P. M. Hansen, C. Lee and D. A. Patterson, VLSI Multiprocessor/Memory Interactions for Scientific Computation, Accepted for publication. *Journal of Parallel and Distributed Computing*, 1988.

[Boss87] P. W. Bosshart, C. R. Hewes, M. D. Ales, M. Chang, K. K. Chau, K. Fasham, C. C. Hoac, T. W. Houston, V. Kalyan, S. L. Lusky, S. S. Mahant-Shetti, D. J. Matzke, K. N. Ruparel, J. F. Sexton, C. Shaw, T. Sridhar, D. Stark and A. L. Lee, A 533K-Transistor LISP Processor Chip, *IEEE Journal of Solid State Circuits*, Vol. SC-22, No. 5, October 1987, pp. 808-819.

[Brad85] M. J. Brady and A. Davidson, Flip-chip Bonding With Solder Dipping, *The Review of Scientific Instruments*, Vol. 56, July 1985, pp. 1459-1460, American Institute of Physics.

[Brig74] E. O. Brigham, *The Fast Fourier Transform*, Prentice Hall, Englewood Cliffs, NJ, 1974.

[Brig88] E. O. Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Brod85] J. R. Broddi, E. M. Fields, C. J. Garen, W. P. Hays and J. Tow, WE DSP32 Information Manual, January 25, 1985.

[Brow72] D. T. Brown, R. L. Eibsen and C. A. Thorn, Channel and Direct Access Device Architecture, *IBM Systems Journal*, Vol. 11, No. 3, 1972, pp. 186-199.

[Buch87] I. Y. Bucher and M. L. Simmons, A Close Look at Vector Performance of Register-to-Register Vector Computers and a New Model, *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Vol. 15, No. 1, May 11-14, 1987, pp. 39-45.

[Burg84] R. M. Burger, R. K. Cavin III, W. C. Holton and L. W. Sumney, The Impact of ICs on Computer Technology, *Computer*, Vol. 17, No. 10, October 1984, pp. 88-95, IEEE.

[Care85] M. J. Carey, P. M. Hansen and C. D. Thompson, Sorting Records in VLSI in *Algorithmically Specialized Parallel Computers*, L. Snyder, L. H. Jamieson, D. B. Gannon and H. J. Siegel ed., Academic Press, Orlando, Florida, 1985, pp. 27-36. Also appeared in *Proceedings of the Purdue Workshop on Algorithmically-specialized Computer*

Organizations, West Lafayette, Indiana, October 1982.

[Caso86] A. Casotto, F. Romeo and A. Sangiovanni-Vincentelli, A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells, *ICCAD*, 1986.

[Cass84] B. A. Cassel, ed., *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Chab85] D. M. Chabries, Unpublished data from personal communication, Brigham Young University EE Department, April 1985. Experience using the CSPI MAP200 attached processor.

[Char81] A. E. Charlesworth, An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family, *Computer*, Vol. 14, No. 9, September 1981, pp. 18-27.

[Char86] A. E. Charlesworth and J. L. Gustafson, Introducing Replicated VLSI to Supercomputing, *IEEE Computer*, Vol. 19, No. 3, March 1986, pp. 10-23.

[Chat87] S. Chatterjee and A. Fischer, *A Coprocessor Design Environment*, International Workshop on Hardware Accelerators, Oxford, UK., October 1987.

[Chen85] C. Chen, S. Kong, D. Lee and T. Stetzler, Design Notes for the SPUR Processor in *Proc. of CS292i: Implementation of VLSI Systems*, R. H. Katz ed., University of California, Berkeley, CA, September 1985.

[Chen86] P. Y. Cheng, *The Pixel Planner for the Autonomous Vehicle Test Bed (AVTB)*, Central Engineering Laboratories, FMC Corporation, June 12, 1986.

[Ches88] G. Chesson, XTP/PE (Protocol Engine) Overview, *IEEE Local Area Network Conference*, Minneapolis, MN, October 1988.

[Clar82] J. H. Clark, The Geometry Engine: A VLSI Geometry System for Graphics, *ACM Siggraph Conference Proceedings*, Vol. 16, No. 3, July 1982, pp. 127-133.

[Clar86a] D. W. Clark, Unpublished data from CS 298 Systems Seminar, University of California, October 16, 1986.

[Clar86b] W. A. Clark, Personal Letter, May 20, 1986.

[Cody84] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hansen, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris and D. Stevenson, A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic, *IEEE Micro*, Vol. 4, No. 4, August 1984.

[Cohl81] E. U. Cohler and J. E. Storer, Functionally Parallel Architecture for Array Processors, *Computer*, Vol. 14, No. 9, September 1981, pp. 28-36.

[Cole83] C. T. Cole, Attaching An Array Processor In the UNIX Environment, Master's Report, University of California, Berkeley, CA, April 1983.

[Cole85] C. T. Cole, Unpublished data from personal communication, April 1985.

[Cool65] J. W. Cooley and J. W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series, *Mathematics of Computation*, Vol. 19, No. 90, 1965, pp. 297-301.

[Corb88] R. Corbett, Unpublished data from personal communication, 1988.

[Cray86] *Cray-1 X-MP Hardware Reference Manual*, Cray Research, Inc., 1986. Order No. HR-0097.

[Curn76] H. J. Curnow and B. A. Wichmann, A Synthetic Benchmark, *The Computer Journal*, Vol. 19, No. 1, February 1976.

[Curr83] T. W. Curry and A. Kukhopadhyay, Realization of Efficient Non-numeric Operations Through VLSI, *VLSI '83*, 1983.

[Denn83] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983.

[Digi76] *PDP 11/70 Processor Handbook*, Digital Equipment Corporation, Maynard, MA, 1976.

[Digi81] *VAX Hardware Handbook*, Digital Equipment Corporation, Maynard, MA, 1980-81.

[Digi85] *MicroVAX 78132 Floating-point Unit Data Sheet (Preliminary)*, Digital Equipment Corporation, April 1985.

[Digi87] *U.S. Price List*, Digital Equipment Corporation, 1987.

[Dijk59] E. W. Dijkstra, A Note on Two Problems in Connexion with Graphs, *Numerishe Mathematik*, Vol. 1, 1959, pp. 269-271.

[Ditz80] D. R. Ditzel and D. A. Patterson, Retrospective on High-Level Language Computer Architecture, *Proceedings of the 7th International Symposium on Computer Architecture*, May 1980, pp. 97-104.

[Ditz81] D. R. Ditzel, Reflections on the High-Level Language Symbol Computer System, *IEEE Computer*, Vol. 14, No. 7, July 1981, pp. 55-66.

[Dobr87] T. Dobry, A High-Performance Architecture for Prolog, Computer Science Division (EECS) Report No. UCB/CSD 87/352, University of California, Berkeley, CA, May 1987. Ph.D. Dissertation.

[Dong79] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1979.

[Dong87] J. J. Dongarra, J. L. Martin and J. Worlton, Computer Benchmarking: Paths and Pitfalls, *IEEE Spectrum*, Vol. 23, No. 7, July 1987, pp. 38-43.

[Dong88] J. J. Dongarra, Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment, *Computer Architecture News*, Vol. 16, No. 1, March 1988, pp. 47-69, ACM Press.

[Elec87] *ATT Sells Solder-bump*, Electronics, October 1987.

[Elec85] *Motorola unveils floating-point coprocessor for 68020*, Electronics Week, January 21, 1985.

[Ewer27] H. H. Ewers, The Sorcerer's Apprentice, also the title of a musical interpretation of Johann Wolfgang von Goethe's ballad, *Der Zauberlehrling*, by Paul Dukas, 1927.

[Fagg77] F. Faggin, Keynote Address: Future Directions in Computer Architecture, *ACM Sigarch Workshop*, Austin, Texas, 1977, pp. 612-614.

[Fand85] J. Fandrianto and B. Y. Woo, VLSI Floating-point Processors, *IEEE Computer Society Press*, May 1985, pp. 93-100.

[Faro83] R. T. Farouki, S. L. Shapiro and S. A. Teukolsky, Computational Astrophysics on the Array Processor, *Computer*, Vol. 16, No. 6, June 1983, pp. 13-15.

[Fauc86] R. Faucette, SPUR Performance Monitor Coprocessor (PMC), Computer Science Division (EECS) University of California, Berkeley, CA, September 30, 1986. Unpublished manuscript.

[Flyn72] M. J. Flynn, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, Vol. C-21, No. 8, September 1972, pp. 948-960.

[Foss88] T. Fossum, Unpublished data from personal communication, Digital Equipment Corporation - High Performance Systems Division, November 1988.

[Fuss84] D. Fuss and C. G. Tull, Centralized Supercomputer Support for Magnetic Fusion Energy Research, *Proceedings of the IEEE*, Vol. 72, No. 1, January 1984, pp. 32-41.

[Geor87] C. J. Georgiou, S. L. Palmer and P. L. Rosenfeld, An Experimental Coprocessor for Implementing Persistent Objects on an IBM 4381, *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 10, No. 2, October 5-8, 1987, pp. 84-87, IEEE Computer Society Press.

[Gibs70] J. C. Gibson, The Gibson Mix, *IBM Systems Development Division Tech. Report*, June 1970.

[Gibs87] G. Gibson, Estimating Performance of Single Bus, Shared Memory Multiprocessors, Computer Science Division (EECS) Report No. UCB/CSD 87/355, University of California, Berkeley, CA, 1987.

[Goer88] R. Goering, Silicon Compilation Boosts Productivity in 88000 Design, *Computer Design*, Vol. 27, No. 9, May 1, 1988, pp. 28.

[Gust88] J. L. Gustafson, Reevaluating Amdahl's Law, *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 532-533.

[Hans83] P. M. Hansen, R. N. Mayo, M. Murphy, M. A. Linton and D. A. Patterson, A Performance Evaluation of the Intel iAPX432 in *Advanced Microprocessors*, A. Gupta and H. D. Toong ed., IEEE Press, June 1983. (Originally appeared as "A Performance Evaluation of the Intel iAPX 432," *Computer Architecture News*, ACM, June 1982, pp. 17-26. Vol. 10, No. 4.).

[Hans85] P. M. Hansen and D. A. Patterson, A Graphical Method for Comparing Computer System Performance Using Floating-point Benchmarks, University of California, Computer Science Division (EECS), Berkeley, CA, November 1985. Unpublished manuscript.

[Hans86] P. M. Hansen and S. I. Kong, SPUR Coprocessor Interface Description, Computer Science Division (EECS) Report No. UCB/CSD 87/308, University of California, Berkeley, CA, October 1986.

[Hans87] J. Hansen, *Architecture For a High Speed Path Optimization Processor*, Central Engineering Laboratories, FMC Corporation, Santa Clara, CA, July 1987.

[Hans88] P. M. Hansen, Coprocessor Architectures for VLSI, Computer Science Division (EECS) Report No. UCB/CSD 88/466, University of California, Berkeley, CA, November 1988. Ph.D. Dissertation.

[Harr85] P. Harris, P. Wensley and E. Wogsberg, Unpublished data from personal communication, Jupiter Computer Corp, April 1985.

[Hear88] *1988 IC Master*, Hearst Business Communications, Garden City, NY, 1988.

[Henn82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, Hardware/Software Tradeoffs for Increased Performance, *Symposium on Architectural Support for Programming Languages and Operating Systems*, Vol. 10, No. 2, March 1-3, 1982, pp. 2-11, IEEE SIGARCH.

[Henn85] J. L. Hennessy, VLSI RISC Processors, *VLSI Systems Design*, Vol. 6, No. 10, October 1985, pp. 22-32.

[Henr88] R. R. Henry, Unpublished data from personal communication, June 1988.

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, Design Decisions in SPUR, *IEEE Computer*, Vol. 19, No. 11, November 1986.

[Hill85] W. D. Hillis, The Connection Machine, Ph.D. Dissertation, MIT Press, May 1985.

[Hock81] R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, 1981.

[Honi84] M. L. Honig and D. G. Messerschmitt, *Adaptive filters: Structures, Algorithms, and Applications*, Kluwer Academic Publishers, 1984.

[Inte81a] in *iAPX 86,88 User's Manual*, Intel Corporation, Santa Clara, CA, 1981, pp. S-3.

[Inte81b] *iAPX 86,88 User's Manual*, Intel Corporation, Santa Clara, CA, 1981.

[Inte85a] 8087 Numeric Data Coprocessor in *Microsystem Components Handbook, Volume I*, Intel Corporation, Santa Clara, CA, 1985, pp. 3.175-3.197.

[Inte85b] 80287 80-Bit HMOS Numeric Processor Extension in *Microsystem Components Handbook, Volume I*, Intel Corporation, Santa Clara, CA, 1985, pp. 4.54-4.78.

[Inte87] *80387 80-Bit CHMOS III Numeric Processor Extension*, Intel Corporation, Santa Clara, CA, January 1987.

[Joup88] N. Jouppi, J. Dion and D. Boggs, *MultiTitan: Four Architecture Papers*, DEC Western Research Lab Report 87/8, April 10, 1988. Paper titles: MultiTitan Central Processor Unit; Multitan Floating-point Unit; MultiTitan Cache Control Unit; MultiTitan Intra-Processor Bus.

[Joy85] W. Joy, Presentation at ISSCC '85 Panel Session, February 1985.

[Kaha85] W. Kahan, Unpublished data from personal communication, April 1985.

[Kais67] J. B. Kaiser, *System Analysis by Digital Computer* , John Wiley and Sons, New York, 1967.

[Kane85] R. Kane, Unpublished data from personal communication, Intel Applications Engineering, April 1985.

[Karp81] W. J. Karplus and D. Cohen, Architectural and Software Issues in the Design and Application of Peripheral Array Processors, *Computer*, Vol. 14, No. 9, September 1981, pp. 11-17.

[Kate83] M. G. H. Katevenis, Reduced Instruction Set Computer Architectures for VLSI, Computer Science Division (EECS) Report No. UCB/CSD 83/141, University of California, Berkeley, CA, October 1983. Ph.D. Dissertation.

[Katz85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon, Implementing a Cache Consistency Protocol, *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, June 1985, pp. 276-283.

[Kava86] R. A. Kavaler, The Design and Evaluation of a Speech Recognition System for Engineering Workstations, Electronics Research Laboratory Memorandum No. UCB/ERL M86/39, University of California, Berkeley, CA, May 5, 1986. Ph.D. Dissertation.

[Kirk83] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, *Science*, Vol. 220, No. 4598, May 1983, pp. 671-680.

[Knut71] D. E. Knuth, An Empirical Study of Fortran Programs, *Software Practice and Experience*, Vol. 1, 1971, pp. 105-133.

[Knut75] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms (Vol 2)*, Addison-Wesley, Menlo Park, California, 1975.

[Koba84] M. Kobayashi, Dynamic Characteristics of Loops, *IEEE Transactions on Computers*, Vol. C-33, No. 2, 1984.

[Kogg81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill and Hemisphere Publishing Company, New York, NY, 1981.

[Kuck78] D. J. Kuck, *The Structure of Computers and Computations Vol. 1*, John Wiley & Sons, New York, NY, 1978.

[Kung81] H. T. Kung and M. Foster, Recognize Regular Languages with Programmable Building-Blocks, *VLSI '81*, August 1981.

[Latt82] W. Lattin, J. R. Rattner and J. Palmer, *Private Communication*, Industrial Liason Program, University of California, Berkeley, CA, March 1982.

[Lawl76] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976.

[Lee86] C. Lee, Micro-architecture of the SPUR Floating-point Unit, University of California, Berkeley, Computer Science Division, Unpublished Report, Berkeley, CA, March 17, 1986.

[Leun86] B. Leung and Y. M. Lin, Statistics on Floating-point Arithmetic, *CS 252 Class Project*, May 1986.

[Lind86] T. Linden, J. Marsh and D. L. Dove, Architecture and Early Experience with Planning for the ALV, *IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 7-10, 1986.

[Mars80] J. P. Marsh and H. L. Steadman, United States Patent No. 4,210,962, July 1, 1980.

[Mash87] K. Mashiko, M. Nagatomo, K. Arimoto, Y. Matsuda, K. Furutani, T. Matsukawa, M. Yamada, T. Yoshihara and T. Nakano, A 4-Mbit DRAM with Folded-Bit-Line Adaptive Sidewall-Isolated Capacitor Cell, *IEEE Journal of Solid-State Circuits*, Vol. SC-22, No. 5, October 1987.

[Mayo86] R. N. Mayo, Unpublished data from personal communication, University of California, Berkeley, Computer Science Division, June 1986. Reports experience with the Magic VLSI layout system.

[McEw85] R. B. McEwen, R. E. Witmer and B. S. Ramey, *USGS Digital Cartographic Data Standards - Digital Elevation Models*, Department of the Interior, United States Geological Survey, 1985.

[McMa86] F. H. McMahon, The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range, UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Menlo Park, California, 1980.

[Melv88] S. Melvin, Unpublished data from personal communication, University of California, Berkeley, CS Division, November 1988.

[Mess87] D. G. Messerschmitt, Unpublished data from personal communication, University of California, Berkeley, EECS Department, March 1987.

[Mira83] G. S. Miranker, L. Tang and C. K. Wong, A Zero Time Sorter, *IBM Journal on Research and Development*, Vol. 27, No. 2, March 1983.

[Mitr85] D. Mitra, F. Romeo and A. Sangiovanni-Vincentelli, Convergence and Finite-time Behavior of Simulated Annealing, *Proceedings of the IEEE Conference on Decision and Control*, 1985.

[Mola88] C. Molar, G. S. Miranker, J. Rubenstein, J. Sanguinetti, R. Allen, B. Borden, S. Johnson, M. Kaplan, W. Ting and C. Wetherell, The Dana Personal Supercomputer (related papers), *Proceedings of the IEEE Computer Society International Conference (COMPCON)*, February 29-March 4, 1988, pp. 448-467.

[Moon85] D. A. Moon, Architecture of the Symbolics 3600, *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, June 1985, pp. 76-83.

[Moto85] *MC68881 Floating-point Coprocessor User's Manual*, Motorola, 1985.

[Moto87] *MC68030 32-Bit Microprocessor User's Manual*, Motorola, 1987.

[Murp64] B. T. Murphy, Cost-Size Optima of Monolithic Integrated Circuits, *Proceedings of the IEEE*, Vol. 54, No. 12, December 1964, pp. 1537-1545.

[Myer78] G. J. Myers, *Advances in Computer Architecture*, John Wiley & Sons, New York, 1978.

[Myer86] G. J. Myers, A. Y. C. Yu and D. L. House, Microprocessor Technology Trends, *Proceedings of the IEEE*, Vol. 74, No. 12, December 1986, pp. 1605-1622, IEEE.

[Nati84] *Series 32000 Databook*, National Semiconductor Corporation, Santa Clara, CA, 1984.

[Nave80] R. Nave and J. Palmer, A Numeric Data Processor, *1980 IEEE International Solid-State Circuits Conference*, February 14, 1980, pp. 108,109.

[Oppe75] A. V. Oppenheim and R. W. Shafer, *Digital Signal Processing*, Prentice-Hall, 1975.

[Ordy83] G. M. Ordy and C. W. Rose, The N.2 System, *ACM IEEE 20th Design Automation Conference*, Miami, FL, June 1983, pp. 520-526.

[Oust88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, The SPRITE Network Operating System, *IEEE Computer*, Vol. 21, No. 2, February 1988, pp. 23-35.

[Palm80] J. Palmer, The Intel 8087 Numeric Data Processor, *Proceedings of the 7th Annual Symposium on Computer Architecture*, 1980, pp. 174-181.

[Paro85] A. M. Parodi, Multi-goal Real-time Global Path Planning for an Autonomous Land Vehicle Using a High-speed Graph Search Processor, *International Conference on Robotics and Automation*, St. Louis, March 25-28, 1985, pp. 161-167.

[Patt85] Y. Patt, Unpublished data from personal communication, April 1985.

[Patt80] D. A. Patterson and D. R. Ditzel, The Case for the Reduced Instruction Set Computer, *Computer Architecture News*, Vol. 8, No. 6, 15 October 1980, pp. 25-33.

[Patt84] D. Patterson, RISC Watch, *Computer Architecture News*, March 1984.

[Ples86] A. R. Pleszkun, G. S. Sohi, B. Z. Kahhaleh and E. S. Davidson, Features of the Structured Memory Access (SMA) Architecture, *Proceedings of the IEEE Computer Society International Conference (COMPCON)*, San Francisco, CA, March 3-6, 1986, pp. 259-263.

[Pren87] *MC68881/MC68882 Floating-point Coprocessor User's Manual*, Prentice-Hall, 1987.

[Radi82] G. Radin, The 801 Minicomputer, *Symposium on Architectural Support for Programming Languages and Operating Systems*, Vol. 10, No. 2, March 1-3, 1982, pp. 39-47, IEEE SIGARCH.

[Robi76] S. K. Robinson and I. S. Torsun, An Empirical Analysis of Fortran Programs, *The Computer Journal*, Vol. 19, No. 1, 1976, pp. 56-62.

[Rome84] F. Romeo and A. Sangiovanni-Vincentelli, Probabilistic Hill Climbing Algorithms: Properties and Applications, *1985 Chapel Hill Conference on VLSI*, and Electronics Research Laboratory Memorandum No. UCB/ERL M84/34, 1984, 1984.

[Rowe88] C. Rowen, M. Johnson and P. Ries, The MIPS R3010 Floating-point Coprocessor, *IEEE Micro*, June 1988, pp. 53-62.

[Ruet86] P. A. Ruetz, Architectures and Design Techniques for Real-time Image Processing ICs, Electronics Research Laboratory Memorandum No. UCB/ERL M86/37, University of California, Berkeley, CA, May 2, 1986. Ph.D. Dissertation.

[Sarr85] C. Sarreno, Unpublished data from personal communication, Applications Engineering, Motorola Advanced Microprocessor Division, April 1985.

[Shan77] R. C. Shank and R. P. Abelson, *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures*, L. Erlbaum Associates, 1977.

[Siew82] D. P. Siewiorek, C. G. Bell and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, NY, 1982.

[Sipp82] T. N. Sippel, Floating RISCs: Implementation and Analysis of Floating-point on RISC I, Computer Science Division (EECS) Unpublished Masters Report, 1982.

[Smit85] J. E. Smith and A. R. Pleszkun, Implementation of Precise Interrupts in Pipelined Processors, *Proceedings of the 12th International Symposium on Computer Architecture*, June 1985, pp. 36-44.

[Sohi87] G. S. Sohi and S. Vajapeyam, Instruction Issue Logic for High-performance Interruptable Pipelined Processors, *Proceedings of the 12th International Symposium on Computer Architecture*, June 1987, pp. 27-34.

[Stec88] R. Steck, Unpublished data from personal communication, Intel Corporation, Beaverton, OR, October 1988.

[Ston87] H. S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, Reading, MA, October 1987.

[Sze85] S. M. Sze, *Semiconductor Devices, Physics and Technology*, John Wiley and Sons, New York, NY, 1985.

[Tayl83] G. S. Taylor, Arithmetic on the ELXSI System 6400, *Proceedings of IEEE 1983 6th Symposium on Computer Arithmetic*, 1983.

[Tayl86] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson and B. G. Zorn, Evaluation of the SPUR Lisp Architecture, *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, June 1986.

[Tekt83] *Digital Analysis System 9100 Series User's Manual*, Tektronix Corporation, Beaverton, OR, 1983.

[Tekt86] *91DVV2 (DAS VLSI Verification - version 2) Installation and User's Manuals*, Tektronix Corporation, Beaverton, OR, 1986.

[Texa76] *The TTL Data Book for Design Engineers, 2nd Edition*, Texas Instruments, Dallas, Texas, 1976.

[Thei83] D. J. Theis, Applications for Array Processors, *Computer*, Vol. 16, No. 6, June 1983, pp. 13-15.

[Thor77] J. E. Thornton, Parallel Operation in the Control Data 6600, *AFIPS Proceedings, Fall Joint Computer Conference*, 1964, pp. 33-40.

[Tick83] E. Tick and D. H. D. Warren, *Towards a Pipelined Prolog Processor*, Artificial Intelligence Center, SRI International, August 1983.

[Vite79] A. J. Viterbi and J. K. Omura, *Principles of Digital Communications and Coding*, McGraw-Hill, New York, 1979.

[Vlah88] H. Vlahos and V. Milutinovic, GaAS Microprocessors and Digital Systems, *IEEE MICRO*, February 1988, pp. 28-56.

[Warr83] D. H. D. Warren, *An Abstract Prolog Instruction Set*, Artificial Intelligence Center, SRI International, August 1983.

[Wile83] R. Wilensky, *Planning and Understanding - A Computational Approach to Human Reasoning*, Addison-Wesley, 1983.

[Wils88a] R. Wilson, Designers' Buying Guide to Microprocessors and Peripheral ICs, *Computer Design*, Vol. 27, No. 4, February 15, 1988, pp. 77-127.

[Wils88b] R. Wilson, Newest Floating-point Processors Blur Architectural Distinctions, *Computer Design*, Vol. 27, No. 8, April 15, 1988, pp. 32-43.

[Wulf88] W. A. Wulf, The WM Computer Architecture, *Computer Architecture News*, March 1988, pp. 70-84.

[Zilo83] *Z8000 CPU Technical Manual*, Zilog, Inc., Campbell, CA, January 1983.