MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Technical Report 1459 August, 1993

The Named-State Register File

Peter Robert Nuth

The Named-State Register File

Peter Robert Nuth

Bachelor of Science in Electrical Engineering and Computer Science, MIT 1983
Master of Science in Electrical Engineering and Computer Science, MIT 1987
Electrical Engineer, MIT 1987

Submitted to Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, August 1993

The research described in this report was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0783 and F19628-92C-0045, and by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

The Named-State Register File

Peter Robert Nuth

Submitted to the Department of Electrical Engineering and Computer Science on
August 31, 1993
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

A register file is a critical resource of modern processors. Most hardware and software mechanisms to manage registers across procedure calls do not efficiently support multithreaded programs. To switch between parallel threads, a conventional processor must spill and reload thread contexts from registers to memory. If context switches are frequent and unpredictable, a large fraction of execution time is spent saving and restoring registers.

This thesis introduces the *Named-State Register File*, a fine-grain, fully-associative register organization. The NSF uses hardware and software mechanisms to manage registers among many concurrent activations. The NSF enables both fast context switching and efficient sequential program performance. The NSF holds more live data than conventional register files, and requires much less spill and reload traffic to switch between concurrent active contexts. The NSF speeds execution of some sequential and parallel programs by 9% to 17% over alternative register file organizations. The access time of the NSF is only 6% greater than a conventional register file. The NSF adds less than 5% to the area of a typical processor chip.

This thesis describes the structure of the Named-State Register File, evaluates the cost of its implementation and its benefits for efficient context switching. The thesis demonstrates a prototype implementation of the NSF and estimates the access time and chip area required for different NSF organizations. Detailed architectural simulations running large sequential and parallel application programs were used to evaluate the effect of the NSF on register usage, register traffic, and execution time.

Keywords: multithreaded, register, thread, context switch, fully-associative, parallel processor.

Author: Peter R. Nuth
nuth@ai.mit.edu

Supervisor: William J. Dally
Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Looking back at almost half a lifetime spent at MIT, it is impossible to thank all of the individuals who have helped me over the years. I am grateful to have been able to work with and learn from so many truly exceptional people.

My thanks first and foremost to Professor Bill Dally, who has taught me more about computer architecture than I would have thought possible. Bill's drive, enthusiasm, and technical ability encouraged me during my time in the CVA group. Bill created an exciting environment, gave us the opportunity to work on challenging projects, and stayed committed to them to the end.

I would like to thank my readers, Professors Tom Knight and Greg Papadopoulos, for their insight and perspective which greatly improved the quality of this thesis. I thank Dr. Bert Halstead for his gentle teachings and crucial support. He introduced me to parallel processors, and first started me on this line of research.

Mike Noakes has been a source of wit and wisdom, a thoughtful critic of all poorly formed ideas, and an ally against the perversity of the modern world. Stuart Fiske gave me good advice, encouragement, and constant reminders not to take myself too seriously. Steve Keckler endured several drafts of this thesis with good humor and an uncompromising pen.

David Harris build the prototype chip in one summer, in an impressive display of skill and single minded devotion. Theodore Tonchev wrote the first version of the TL0 translator.

I would like to thank my family for their constant encouragement and faith in my ability. Although we have pursued different paths, my sisters supported my decisions, and applauded every achievement. I especially thank my parents, whose love and discipline instilled me with a determined nature and a sense of curiosity. They helped nurture my dreams, and gave me the rare opportunity to pursue them.

Most of all, I would like to thank my wife, Julia. She has been a caring friend and a tireless supporter. She has taught me more than any teacher, by teaching me about myself. Through her I have found commitment, love, and great joy.

This thesis is dedicated to my parents,
Robert and Lois Nuth

Table of Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

1.1 Overview

1.1.1 Problem Statement

Most sequential and parallel programming models divide an application into *procedures* that invoke other procedures in a data dependent manner. Each procedure *activation* requires a small amount of run-time state for local variables. While some of this local state may reside in memory, the rest occupies the processor's register file. The register file is a critical resource in most modern processors [34,66]. Operating on local data in registers rather than memory speeds access to that data, and allows a short instruction to access several operands [75,28].

There have been many proposals for hardware and software mechanisms to manage the register file and to efficiently switch between activations [67,86]. These techniques work well when the activation sequence is known, but behave poorly if the order of activations is unpredictable [42]. Dynamic parallel programs [60,21,39,52], in which a processor may switch between many concurrent activations, or *threads*, run particularly inefficiently on conventional processors. To switch between parallel threads, conventional processors must spill a thread's *context* from the processor registers to memory, then load a new context. This may take hundreds of cycles [36]. If context switches are frequent and unpredictable, a large fraction of execution time is spent saving and restoring registers.

1.1.2 Proposal

The thesis introduces the *Named-State Register File*, a register file organization that permits fast switching among many concurrent activations while making efficient use of register space. It does this without sacrificing sequential thread performance, and can often run sequential programs more efficiently than conventional register files.

The NSF does not significantly increase register file access time. While the NSF requires more chip area per bit than conventional register files, that storage is used more effectively, leading to significant performance improvements over alternative register files.

The goals of this research are:

- To reduce the cost of context switching by reducing the frequency and number of registers which must be saved and restored from memory.

- To make more effective use of processor registers, which are the most critical memory in a computer system.
- To run both sequential and parallel code efficiently.

1.1.3 Methodology

This thesis describes the structure of the Named-State Register File, evaluates the cost of its implementation, and its benefits for efficient context switching. A prototype chip was built to estimate the access time and VLSI chip area required for several different NSF organizations. Detailed architectural simulations running large sequential and parallel application programs are used to evaluate the effect of the NSF on register usage, register traffic, and execution time.

1.1.4 Research Results

- The NSF holds more active data than a conventional register file with the same number of registers. For the large sequential and parallel applications tested, the NSF holds 30% to 200% more active data than an equivalent register file.
- The NSF holds more concurrent active contexts than conventional files of the same size. The NSF holds 20% more contexts while running parallel applications. For sequential programs, the NSF holds twice as many procedure call frames as a conventional file.
- The NSF is able to support more resident contexts with less register spill and reload traffic. The NSF can hold the entire call chain of a large sequential application, spilling registers at 10^{-4} the rate of a conventional file. On parallel applications, the NSF reloads 10% as many registers as a conventional file.
- The NSF speeds execution of sequential applications by 9% to 18%, and parallel applications by 17% to 35%, by eliminating register spills and reloads.
- The NSF's access time is only 5% greater than conventional register file designs. This should have no effect on processor cycle time.
- The NSF requires 30% to 50% more chip area to build than a conventional file. This amounts to less than 5% of a typical processor's chip area.

1.2 Justification

This thesis makes several assumptions about the behavior of parallel programs:

- Parallel *threads* are spawned dynamically by the program.
- Parallel programs contain phases of high parallelism and of sequential critical paths.

- Parallel processors must frequently switch threads to avoid idling during long communication and synchronization delays.
- Parallel thread schedulers must balance parallelism and locality.

The next few sections illustrate these assumptions and the effect they have on processor architecture.

1.2.1 Parallel and Sequential Programs

The NSF is designed to execute both parallel and sequential programs efficiently. Figure 1-1 shows example call graphs of sequential and parallel applications. Both are dynamic models of computation, in which a procedure may invoke one of several other procedures, depending on program data. In the sequential program, only one sequential procedure may be running at a time, and the call chain can be allocated on a stack.

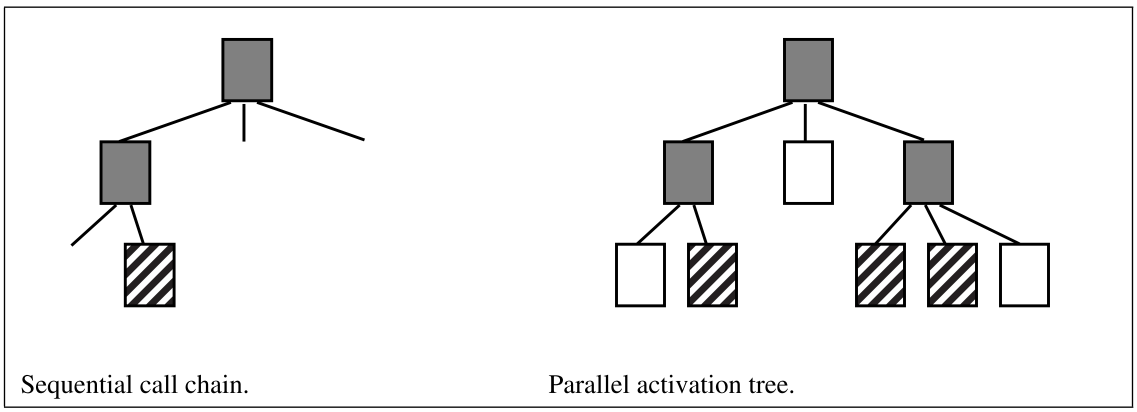


FIGURE 1-1. Call graphs of sequential and parallel programs.
Shaded *activations* are waiting at synchronization points, and cannot proceed. Cross-hatched activations are running on some processor. Only one sequential procedure activation can run at a time. Many parallel *threads* may be ready to run at a time, and a processor may interleave execution of several threads.

The parallel program, on the other hand, may dynamically spawn parallel procedure invocations, or *threads*[1]. A parent may be able to spawn a number of child threads before having to wait for a result to be returned [60]. Since several threads may be able to run at the same time, the activation tree is heap allocated. Threads may also interact through shared variables [9] or message passing [39].

Since parallel threads are spawned dynamically in this model, and threads may synchronize with other processors in a data-dependent manner, the order in which threads are

1. Contrast with other models of parallelism, in which the number of parallel tasks is fixed at compile time [29].

executed on a single processor of a parallel computer cannot be determined in advance. A compiler may be able to schedule the execution order of a local group of threads [42], but in general will not be able to determine a total ordering of threads across all processors.

1.2.2 Parallel Program Behavior

While most dynamic parallel programs generate significant numbers of parallel tasks, that parallelism is not sustained during the entire lifetime of the program. Figure 1-2 shows the parallelism profile [45] for a timestep of Simple(50), a typical Dataflow application [60]. The initial phase of the program produces an extremely large number of parallel tasks, but the program ends in a long tail with low parallelism, to merge the results of the computation. Speeding up that sequential critical path is as important as exploiting the total parallelism of the application. For this reason, processors for parallel computers must be able to efficiently run both parallel and sequential code.

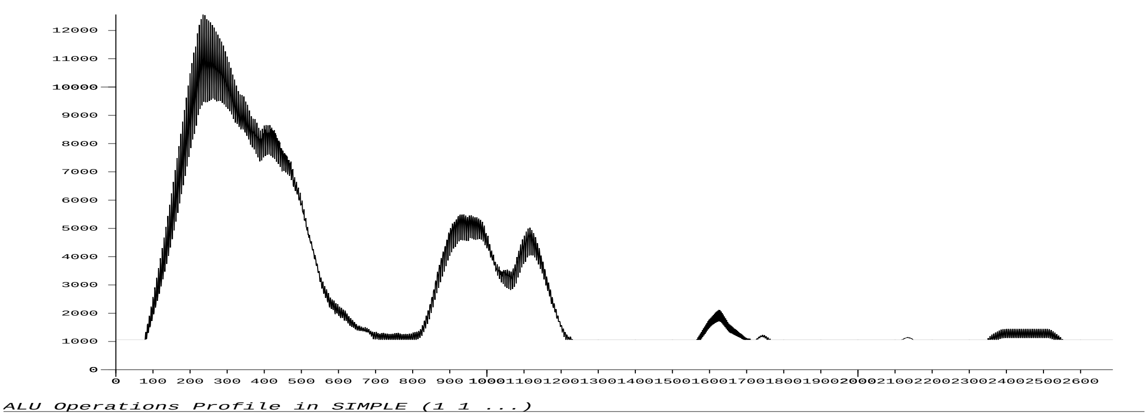


FIGURE 1-2. Parallelism profile of a Simple(50), a typical dataflow application. The program consists of phases of very high parallelism, as well as long sequential tails.

1.2.3 Context Switching

In spite of the large amount of parallelism available in many applications, there are several problems in running programs on large scale multicomputer systems [8].

The first problem is that most applications must pass data between physically separate components of a parallel computer system. As ever larger systems are built, the time required to communicate across the computer network increases. This communication latency has not kept pace with decreasing processor cycle times. Even very low-latency networks [62,83] have round trip message latencies greater than 100 instruction cycles. Fine grain programs send messages every 75 to 100 instructions [38,15]. If processors must wait for each message to complete, they will spend an ever increasing amount of time idle.

Another problem is synchronization between threads, since highly parallel programs consist of short threads that frequently exchange data. The average run length of such a thread between synchronization points may be 20 to 80 instructions [21]. Each synchonization point may require an unbounded amount of time to resolve [55]. Stalling at every synchronization point would waste a large fraction of the processor's performance.

Figure 1-3 illustrates one alternative to idling a processor on communication and synchronization points. Efficient context switching allows a processor to very quickly switch to another thread and continue running.

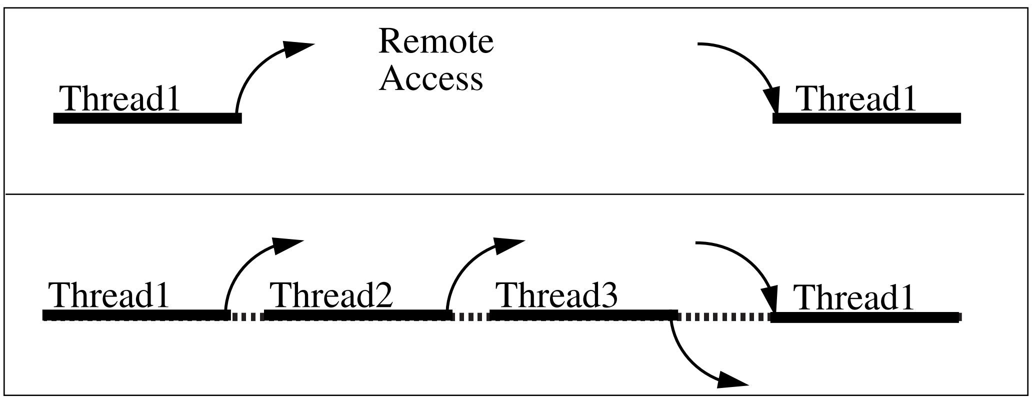


FIGURE 1-3. Advantage of fast context switching.
A processor idling on remote accesses or synchronization points (top), compared with rapid context switching between threads (bottom).

The less time spent context switching, the greater a processor's utilization [1]. Equation 1-1 shows the utilization of a processor as a function of average context switch time T_{switch}, and average run length of threads, T_{run}, assuming enough concurrent threads.

$$M_{thread} = \frac{T_{run}}{T_{run} + T_{switch}} \quad \text{(EQ 1-1)}$$

1.2.4 Thread Scheduling

Scheduling threads to run in parallel computer systems is an active area of research. This thesis makes no specific assumptions about the order in which threads are run. However, most dynamic thread scheduling algorithms must balance parallelism against resources.

As illustrated by Figure 1-2, dynamic parallel programs may generate excessive parallelism. Since each active thread consumes memory, many thread scheduling policies must limit the number of concurrent active threads [20,38,48]. The goal is to spawn enough parallelism to keep processors busy without completely swamping the system.

A second goal of thread scheduling is to exploit temporal locality among threads. An efficient algorithm [55] for switching threads in response to synchronization delays is to switch between a small number of active threads. If a thread halts on a synchronization point, the processor switches to another thread in the active set. A thread is only swapped out of the active set if it has been stalled for a long time. Then another thread from outside the active set is loaded in it place. This helps ensure temporal locality in the execution of the active threads. This scheduling policy can be extended to several more levels, such that the processor gives preference to threads in the inner set, but must occasionally load a thread from outer sets in order to make progress [61].

The sections that follow outline different solutions to the problem of efficient context switching. The Named-State Register File is proposed as an alternative register file organization that meets the goals of fast context switching, good sequential performance, and efficient use of processor resources.

1.3 Multithreaded Processors

Multithreaded processors [19,76,82,29] reduce context switch time by holding the state of several threads in the processor's high speed memory. Typically, a multithreaded processor divides its local registers among several concurrent threads. This allows the processor to quickly switch among those threads, although switching outside of that small set is no faster than on a conventional processor.

Multithreaded processors may interleave successive instructions from different threads on a cycle-by-cycle basis [76,47,65,54]. This prevents pipeline bubbles due to data dependencies between instructions, or long memory latencies. Other processors interleave blocks of instructions from each concurrent thread [29,4,23]. This exploits conventional processor pipelines, and performs well when there is insufficient parallelism. While the techniques introduced in this research are equally applicable to both forms of multithreading, we will usually discuss them in terms of block interleaving.

1.3.1 Segmented Register Files

Figure 1-4 describes a typical implementation of a multithreaded processor [76, 47,4,5]. This processor partitions a large register set among a small set of concurrent threads. Each register *frame* holds the registers of a different thread. A *frame pointer* selects the current active frame. Instructions from the current thread refer to registers using short offsets from the frame pointer.

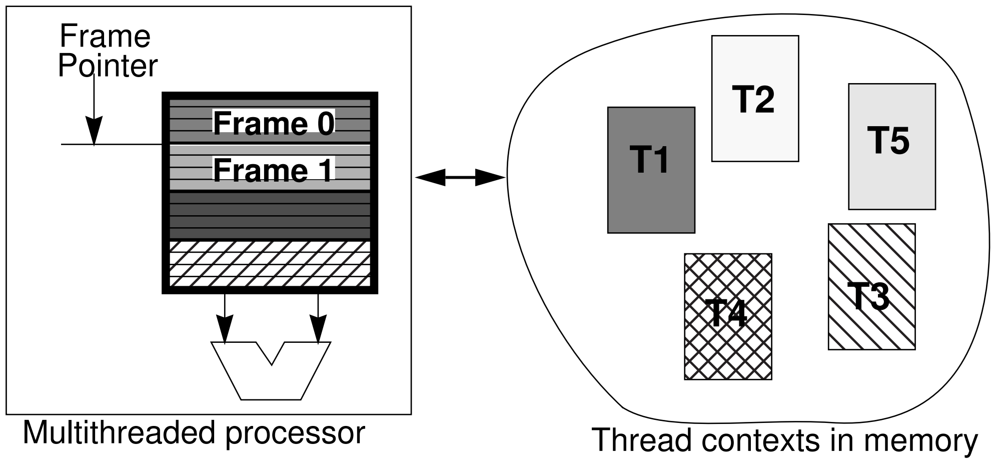


FIGURE 1-4. A multithreaded processor using a segmented register file.
The register file is segmented into equal sized frames, one for each concurrent thread. The processor spills and restores thread contexts from register frames into main memory.

Switching between the resident threads is very fast, since it only requires setting the frame pointer. However, often none of these resident threads will be able to make progress, and the processor must switch to a thread outside of this small set. To switch to a non-resident thread, the processor must spill the contents of a register frame out to memory, and load the registers of a new thread in its place.

This static partitioning of the register file is an inefficient use of processor resources. In order to load a new context, the processor must spill and reload an entire register frame. Some threads may not use all the registers in a frame. Also, if the processor switches contexts frequently, it may not access all the registers in a context before it must spill them out to memory again. In both cases, the processor will waste memory bandwidth loading and storing registers that are not needed.

Dividing the register file into large, fixed sized frames also wastes space in the register file. At any time, some fraction of each register frame holds *live* variables, data that will soon be accessed by an instruction. The remainder of the frame's registers are not used. Since each thread is allocated the same fixed sized frame, many registers in the register file will contain dead or unused variables. This is a serious inefficiency, since the register file, as the memory nearest to the processor's ALU, is the most precious real-estate in the machine. A more efficient scheme would hold only current, live data in the register file.

As noted in Section 1.2, communication latencies in parallel computers may be long and variable, and synchronization delays may be frequent and unbounded. As larger parallel computers are built, processors require more active threads to stay busy. A processor for such a system will spend considerable time spilling and reloading threads.

However, since all parallel programs contain sequential critical paths, a processor must also be able to run sequential code efficiently. Devoting large amounts of chip area to holding many active frames may be an inefficient use of the processor chip. And many highly multithreaded machines cannot run sequential code efficiently [76,47].

The fundamental problem with this segmented register file organization is the same as with conventional register files. The processor is binding a set of variable names (for a thread) to an entire block of registers (a frame). A more efficient organization would bind variable names to registers at a finer granularity.

1.4 The Named-State Register File

The *Named-State Register File* (NSF) is an alternative register file organization. It is not divided into large frames for each thread. Instead, a thread's registers may be distributed anywhere in the register array, not necessarily in one continuous block. An active thread may have all of its registers resident in the array, or none. The NSF dynamically allocates the register set among the active threads.

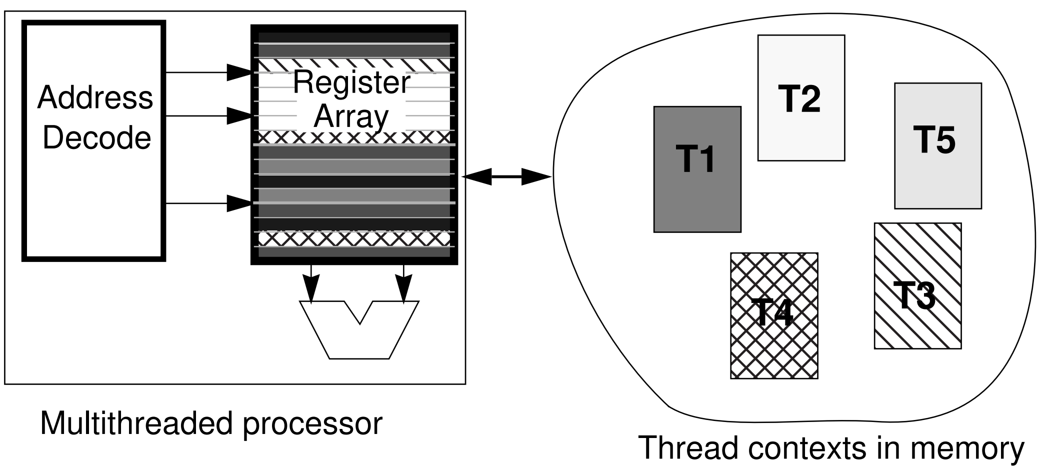


FIGURE 1-5. A multithreaded processor using a Named-State Register File.
The NSF may hold registers from a number of different contexts resident simultaneously. The processor spills and restores individual registers to main memory as needed by the active threads.

As shown in Figure 1-5, the NSF may hold data from a large number of contexts resident simultaneously. The NSF does not explicitly flush and reload contexts after a thread switch. Registers are loaded on demand by the new thread. Registers are only spilled out of the NSF as needed to clear space in the register file.

The NSF allows a processor to interleave many more threads than segmented files, since it sets no fixed limit on the number of resident threads. The NSF keeps more active data resident than segmented files, since it is not coarsely fragmented among threads. It spills and reloads far fewer registers than segmented files, since it only loads registers as they are needed.

Instructions refer to registers in the NSF using a short offset, just as in conventional register files. However, rather than using a frame pointer to distinguish between thread registers, the NSF assigns a unique *Context ID* to each concurrently running thread. Individual registers in the NSF are addressed using both Context ID and register offset. This allows registers from different threads mix freely in the register file.

The NSF uses hardware and software mechanisms to dynamically bind registers in the register file. The NSF is a fully-associative structure, with very small lines. In this way, the NSF binds variable names to individual registers.

Chapter 2 describes the structure and operation of the Named-State Register File in more detail. It compares the NSF to conventional register files, and discusses the advantages of different memory structures.

1.5 Related Work

1.5.1 Memory to Memory Architectures

Much of the motivation for this research originated with Iannucci's work on Hybrid Dataflow [42]. The compilation techniques he introduced for grouping dataflow instructions into sequential threads for execution on conventional processors were also adopted by the Berkeley TAM project [21]. But while Hybrid Dataflow used a processor's registers for temporary variables, it was unable to keep any register state across context switch boundaries. Even *potential* context switch points forced registers to be flushed to memory. Given the short thread run lengths involved, this was clearly an inefficient use of processor resources.

The *Monsoon* processor [65], on the other hand, explicitly switched contexts in response to each Dataflow token, potentially on every instruction. Monsoon used no general registers, but fetched operands from a large, high speed *local storage*. Monsoon compensated for the slower operand access time by using a very deep processor pipeline.

Omondi [63] has proposed a memory-accumulator processor to support multiple concurrent tasks. The machine fetched operands directly from the data cache. This organization also used a very deep processor pipeline to cope with long operand fetch latencies.

Each of these machines avoided storing operands in registers as a way of speeding up context switching. Yet, as noted by Sites [75], and Goodman [28], registers are a more effective use of on-chip area than data caches. Register files are faster and easier to multi-port than caches. Instructions can directly address a register set, but not the entire memory space mapped into a data cache. And finally, compilers are able to use register allocation techniques [16] to effectively use that precious real-estate. Chapter 2 revisits these issues in more detail.

1.5.2 Preloading

Arvind [59] and Agarwal [77] have proposed register file organizations that either pre-load contexts before a task switch, or spill contexts in the background. Usually this requires extra register file ports, and some control logic. While these techniques speed context switching, since the target context has already been preloaded, they do not make efficient use of processor resources. Neither technique reduces the register spill and reload traffic of conventional multithreaded register files, but only re-schedules it. These techniques also use processor space inefficiently, since only a fraction of the register file contains live data.

1.5.3 Software Techniques

Several software techniques have been proposed to run multithreaded programs more efficiently on existing hardware. Keppel [49] and Hidaka [36] both proposed different ways of running multiple concurrent threads in the register windows of a Sparc [14] processor. Both propose variations on standard Sparc window overflow traps. The Sparcle chip built by Agarwal [4] modifies a Sparc chip by adding trap hardware. Sparcle also uses tuned trap handlers to speed context switching. At best, register windows when used in this way are very similar to the segmented register file described in Section 1.3.1, and have the same disadvantages. This large, fixed partitioning leads to poor utilization of the register file, and high register spill traffic.

Waldspurger [85] has proposed small modifications to a processor pipeline, and compiler and runtime software to allow different threads on a multithreaded processor to share the register set. This technique allows each thread to declare the number of registers it will use, so that different threads have different frame sizes in the register file. Runtime software is responsible for dynamically packing these different frame sizes into the register file. Such small register frames can improve the utilization of the register file at the cost of more register to memory traffic. It remains to be seen how well compilers can determine the optimum frame size for a thread, and how well runtime software can allocate those frames. In contrast, the NSF allows a much more dynamic binding of registers to contexts, so that an active thread can use a larger proportion of the register file.

The TAM [21] project takes a different approach in compiling fine-grain parallel Dataflow applications. The TAM compiler groups parallel threads into activations, in order to reduce the number of context switches by running a longer instruction stream between synchronization points. This is a useful technique regardless of the underlying hardware. This thesis uses the TAM compiler to produce parallel code for NSF simulations.

1.5.4 Alternative Register Organizations

There have been many proposals for alternative register file structures to run sequential code. Most of these approaches attempt to minimize register spills and reloads by supporting variable sized procedure activation frames. None are well designed to handle arbitrary switching between multiple concurrent threads, but instead assume that register frames will be allocated and released in strict FIFO order.

Ditzel and McLelland proposed the C-machine [25,11] as a register-less, stack based architecture. The C-machine stores the top of stack in a multiported stack buffer on chip. The processor maps references to the stack into offsets in the stack buffer. Russell and Shaw [70] propose a stack as a register set, using pointers to index into the buffer. These structures might improve performance on sequential code, but are very slow to context switch, because of the implicit FIFO ordering.

Huguet and Lang [40], Miller and Quammen [56], and Kiyohara [51] have each proposed other register file designs that use complex indirection to add additional register blocks to a basic register set. It is not clear if any of these designs are technically feasible. They do not make good use of register area, and may significantly slow down sequential execution.

1.6 Thesis Outline

Chapter 2 describes the structure of the Named-State Register File in more detail. The chapter reviews NSF operation as well as some design alternatives. The chapter compares different memory structures and discusses the benefits of register files for sequential and parallel programs. It concludes with an analysis of the NSF and conventional register files running multiple concurrent threads.

Chapter 3 describes how to build the NSF, and the logic and circuitry required to make it efficient. It outlines circuit simulations that compare the access time of the NSF to conventional register files. It also uses the layout of a prototype chip to determine the area required to build an NSF, relative to a conventional register file.

Chapter 4 outlines the strategy used for simulating the NSF in running real sequential and parallel programs. It describes the software environment, and decisions that were made in modelling performance on the benchmarks.

Chapter 5 describes the results of these software simulations. It compares the utilization and register traffic of NSF and conventional register files. It reveals how this performance scales with the size of the register file. It investigates which factors contribute most to performance. And it computes the overall effect of Named-State on program execution time.

Chapter 6 concludes with a discussion of the costs and benefits of the NSF, and some directions for future research.

CHAPTER 2

The Named-State Register File

This chapter describes the organization and operation of the Named-State Register File. It compares the NSF to other memory structures, and shows that the NSF is effective for managing processor local data. The chapter describes some issues in managing contexts, and alternatives in the design of the NSF. The chapter ends with an analysis comparing the NSF to a conventional register file in running multiple concurrent threads.

2.1 NSF Operation

2.1.1 Structure

Figure 2-1 outlines the structure of the Named-State Register File. The NSF is composed of two components: the register array itself, and a fully-associative address decoder. The NSF is *multi-ported*, as are conventional register files, to allow simultaneous read and write operations. Figure 2-1 shows a three ported register file, that supports two register reads and a write per cycle. While the NSF could be built with more ports, to allow many simultaneous accesses, the remainder of this chapter will concentrate on three ported register files.

Recall that a segmented register file, as described in Section 1.3.1, is composed of several distinct register *frames*. The Named State Register File is instead divided into many short register *lines*. Depending on the design, an NSF line may consist of a single register, or a small set of consecutive registers. Typical register organizations may have line sizes between one and four registers wide.

A conventional register file is a non-associative, *indexed* memory, in that a register address is a physical location, a line number in the register array. Once a register variable has been written to a location in the register file, it does not move until the context is swapped out. Multithreaded register files use *frame pointers* to cycle through the available frames in the array. The *block size* of this register file is an entire frame, since a range of register indices is bound to a frame as a single unit.

The Named-State Register File, on the other hand, is *fully-associative*, since a register address may be assigned to any line of the register file. During the lifetime of a context, a register variable may occupy a number of different locations within the register array. The unit of associativity of the NSF (its block size) is a single line. Each line is allocated or deallocated as a unit from the NSF.

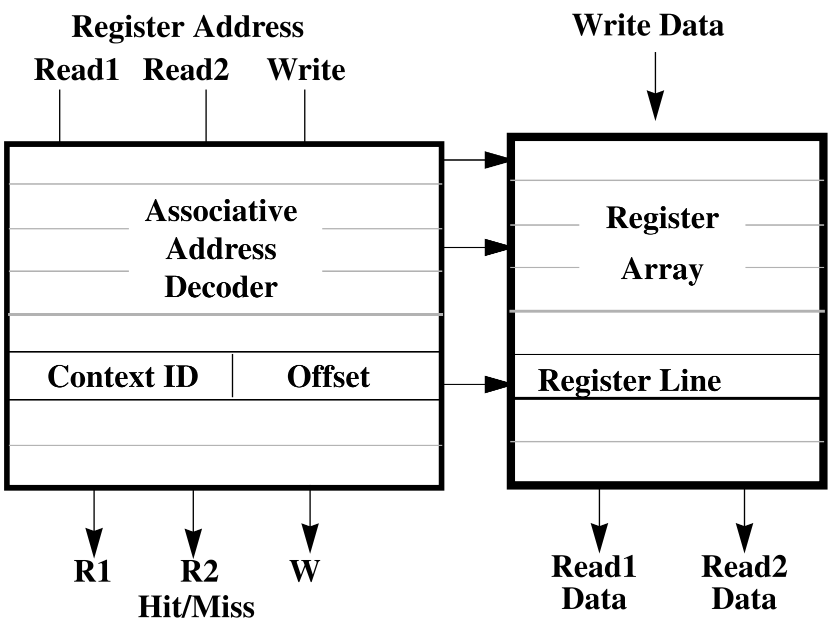


FIGURE 2-1. Structure of the Named-State Register File.

The NSF uses an associative address decoder to achieve this flexibility. The address decoder translates each register address to the line of the array that contains that register. The NSF binds a register name to a location in the register file by programming the address decoder with the register's address. Subsequent reads and writes of that register match the register address against each address programmed into the decoder. Chapter 3 describes the how such an address decoder can be built.

2.1.2 Register addressing

As in any general register architecture, instructions refer to registers in the NSF using a short register *offset*. This identifies the register within the current procedure or thread activation. However, instead of using a *Frame pointer* to identify the current context, the processor tags each context with a *Context ID*. This is a short integer that uniquely identifies the current context from among those resident in the register file.

The NSF does not impose any restrictions on how Context IDs are used by different programming models. In the NSF, each Context ID simply defines a separate, fixed size set of registers. The width of the offset field determines the size of the register set (typically 32 registers). In some programming models, a context may be the state of a procedure activation, or the local state of a single parallel *thread.* Section 2.3 describes some issues related to the management of Context IDs.

A register address in the NSF is the concatenation of its Context ID and offset. The current instruction specifies the register offset, and a processor status word supplies the current CID. In effect, the CID increases the size of the register name space. While a segmented register file may refer to a few (say 4) register frames, the NSF may address the registers of many contexts (say 32 or 64) simultaneously.

While most instructions only refer to registers in the current context, some load and store instructions can copy values from one context to another. A *load_from_context* instruction treats its source operands as the Context ID and offset of a register from another context. The instruction fetches that register and writes it to a register in the current context. A *store_to_context* instruction behaves in a similar manner. This allows a procedure to load arguments from its caller's context, or to return the result of that procedure invocation.

2.1.3 Read, write, allocate and deallocate

Binding an address to a line of the register file is known as *allocating* a register. The first write to a new register allocates that register in the array. Once a register is resident in the array, subsequent reads and writes of that register simply access the appropriate line of the register file.

An instruction may explicitly allocate a register on a write operation. This is a hint from the compiler that the line containing that register has not yet been allocated in the file. Setting this bit in the register address can speed up register allocation, since the NSF does not need to search for that address in the register file before writing the data.

In a similar manner, the NSF can *deallocate* a register after it is no longer needed by the program. A bit in the register read address informs the NSF to read and simultaneously deallocate the register. When all registers in the line have been deallocated, the line is free to be allocated to a new set of register variables.

The NSF can also deallocate all registers associated with a particular context. The *dealloc_CID* instruction will delete any registers from the current context that are resident in the register file. It handles the case where a compiler is unable to determine the last usage of a register in a context. Thus deallocating a register after its last use is not necessary for correct operation of the NSF. It merely makes more efficient use of the register file.

2.1.4 Reloading and spilling

The NSF holds a fixed number of registers. After a register write operation has allocated the last available register line in the register file, the NSF must *spill* a line out of the register file and into memory. The NSF could pick this *victim* to spill based on a number of different strategies. This study simulates a *Least Recently Used* (LRU) strategy, in which the NSF spills from the file the line that has least recently been accessed.

If an instruction attempts to read a register that has already been spilled out of the register file, that read operation will *miss* on that register. The NSF signals a miss to the processor pipeline, stalling the instruction that issued the read. Then the register file *reloads* that register from memory. Depending on the organization of the NSF, it may reload only the register that missed, or the entire line containing that register.

Writes may also miss in the register file. A write miss may cause a line to be reloaded into the file (fetch on write), or may simply allocate a line for that register in the file (write-allocate). Section 2.4 discusses the alternatives in more detail.

This implicit register spilling and reloading is the most significant difference between the NSF and conventional register files. Any instruction may miss in the register file, and require a line to be reloaded. However, the NSF will only reload the line that caused that miss, and not the entire register context. While this strategy may cause several instructions to stall during the lifetime of a context, it ensures that the register file never loads registers that are not needed. As shown in Chapter 5, better utilization of the NSF register file more than compensates for the additional misses on register fetches.

2.1.5 Context switching

Context switching is very fast with the NSF, since no registers must be saved or restored. There is no need to explicitly flush a context out of the register file after a switch. Registers are only spilled or reloaded on demand. After a context switch, the processor simply issues instructions from the new context. These instructions may miss in the register file and reload registers as needed by the new context.

While register allocation and deallocation in the NSF use explicit addressing modes, spilling and reloading are implicit. The instruction stream creates and destroys contexts and local variables, which are known at compile time. The NSF hardware manages register spilling and reloading in response to run-time events. In particular, there are no instructions to flush a register or a context from the register file.

The only concession to thread scheduling routines is the *probe* instruction. This is similar to the *load_from_context* instruction described in Section 2.1.2, but rather than loading the value of a register, it only checks that the register is resident. A *probe* instruction will never cause a line to be reloaded into the register file, nor another line spilled. This allows run-time software to check the status of lines in another context, without disturbing the other contents of the register file. A thread scheduling routine might use *probe* instructions to check if a context has been spilled out of the NSF.

2.2 Justification

The Named-State Register File is an unusual memory structure. This section describes the aspects of the NSF that differentiate it from other memory structures, and how the NSF design allows it to run parallel and sequential code efficiently.

2.2.1 Registers and memory

In papers on the optimal use of processor chip area for local memory, Sites [75] and Patel [27] cite several advantages to allocating that space as registers rather than a cache:

- Registers are typically multi-ported, whereas caches are not.
- Register files are usually small and very fast to access.
- Registers can be identified using short indices in an instruction word. A single instruction may refer to several register operands.
- Registers can be managed by the compiler, which assigns variables to registers based on lifetime analysis within subroutines.

Some of these issues are implementation dependent:

- It is possible to build multi-ported data caches [30].
- The access time of both caches and register files depends on the size and organization.
- Some register-less architectures refer to operands as fixed offsets from a stack pointer [11].

This thesis argues that a critical distinction between caches and register files is in how registers are managed. This results from the difference between register and memory address spaces.

Figure 2-2 illustrates the memory structures used by most modern processors. Note that programs refer to data stored in memory using *virtual memory* addresses. A data or instruction cache transparently captures frequently used data from this virtual address space[1]. In a similar manner, the processor's physical memory stores portions of that virtual address space under control of the operating system.

It is often very difficult for a compiler to manage the virtual address space used for local data [53]. Programs may create aliases to memory locations, index through large structures in a data dependent manner, and dynamically allocate and deallocate memory. Depending on the data set, a particular program may have good or very poor paging and caching performance.

1. The cache may be virtually or physically addressed. A virtual cache is shown here for simplicity.

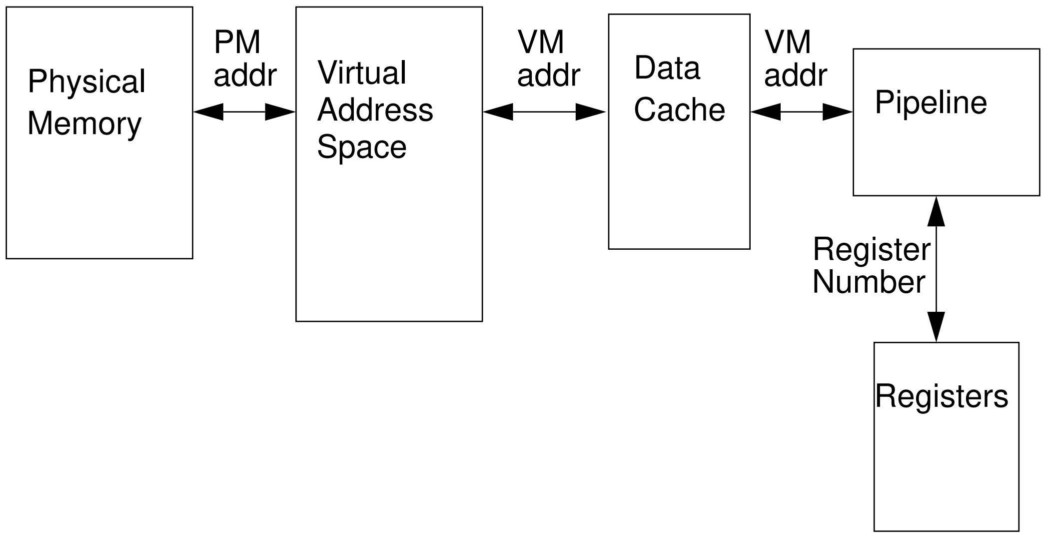


FIGURE 2-2. Memory hierarchy of a modern processor.
The register file is a fast local memory for the processor.
It also defines a register name space distinct from the virtual address space of main memory.

The processor's register file, on the other hand, is not part of this virtual address space. In effect, the register file defines a new name space, addressed only by register number. Since the register set is separate from the rest of memory, a compiler may efficiently manage this space [16], without regard to the data references being made by the program.

Note that a program typically spills and reloads variables from the register set into main memory. This movement from register space to virtual address space is under program control, and a compiler may use local knowledge about variable usage to optimize this movement [78]. A program may use a number of different strategies to hold spilled registers, such as stack and heap frames. When the program determines that a set of variables in the registers are no longer needed, it may overwrite them.

In contrast, a data cache uses a fixed, hardware strategy to capture data from main memory. Both cache and main memory refer to data using the same virtual address. The cache is not the primary home for this data, but must ensure that data is always saved out to memory to avoid inconsistency. Although some caches allow programs to avoid caching some data, or to explicitly allocate data in the cache [68], the program typically has no control over how memory operands are mapped into the cache.

2.2.2 NSF and memory hierarchy

Figure 2-3 illustrates how the Named-State Register File fits into the memory hierarchy. As with conventional register files, the NSF defines a register name space separate from that of main memory. But now the name space consists of a <Context ID: Offset> pair. In effect, the use of Context IDs significantly increases the size of the register name space. Since the NSF is an associative structure, it can hold any registers from this large address space in a small, efficient memory.

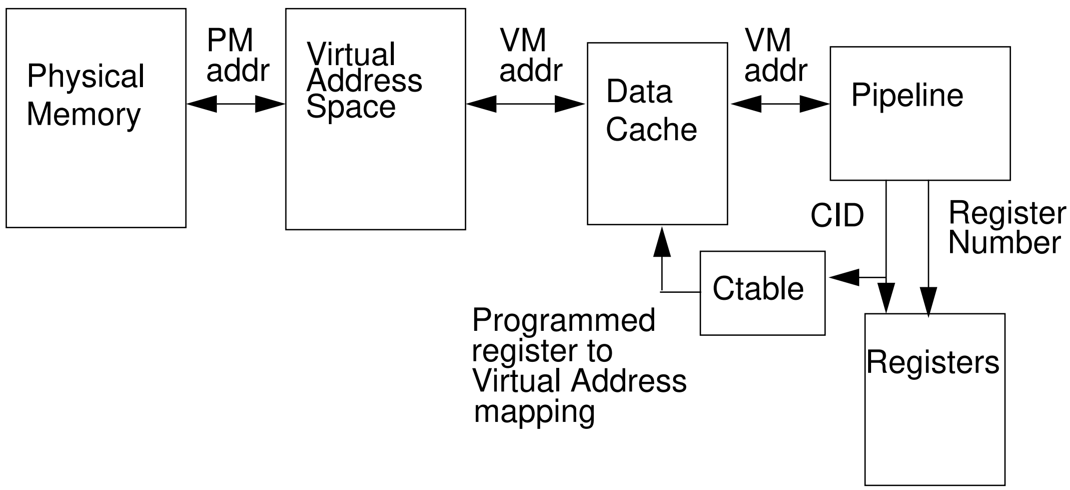


FIGURE 2-3. The Named-State Register File and memory hierarchy.
The NSF addresses registers using a <Context ID: Offset> pair. This defines a large register name space for the NSF. The Ctable is a short indexed table to translate Context IDs to virtual addresses.

The NSF can use the same compiler techniques as conventional register files to effectively manage the register name space. A program may explicitly copy registers to and from the virtual memory space (or backing store) as with conventional register files. But the NSF provides additional hardware to help manage the register file under very dynamic programming models, where compiler management may be less effective [41].

As described in Section 2.1.4, the NSF spills registers to memory when it becomes full. It also reloads registers on demand, as required by instructions from a running program. Figure 2-3 shows how the NSF hardware maps registers into the virtual address space to support spills and reloads. The block labelled *Ctable* is a simple indexed table that translates Context IDs to virtual addresses. This allows the NSF to spill registers directly into the data cache. A user program or thread scheduler may use any strategy for mapping register contexts to structures in memory, simply by writing the translation into the Ctable. This mechanism permits dynamic register spilling by the NSF hardware, under program control.

Note that since CID is a short field, the Ctable can be a short table indexed by CID, rather than an associative lookup. Both read and write operand CIDs are translated through the Ctable in the pipeline stage before operand fetch, in case either operand misses in the NSF.

If a register must be spilled out of the NSF, its CID is translated through the Ctable so that the register line can be written directly into the data cache.

2.2.3 Properties of memory structures

Table 2-1 enumerates different properties of data caches, conventional register files, and the Named-State Register File. A processor architect may use one of these alternatives as fast local memory for temporary data.

Property	Determines	Data Cache	Register File	Named-State File
Latency	Performance	≥1 cycle	< 1 cycle	< 1 cycle
Bandwidth	Performance	1-2 words/cycle	≥ 3 words/cycle	≥ 3 words/cycle
Selection	Performance	Associative	Direct select	Fully-associative
Contents	Program	All program data	Register variables	Register variables
Name Space	ISA	Virtual Address	Register name	Register name
Addressing	ISA	VM/PM address	Short index	CID + index
Sequential Allocation	ISA	Hardware / *Compiled*	Compiled	Compiled/ Hardware
Sequential Management	Program	Hardware	Compiled preload	Hardware/ Compiled
Parallel Management	Program	Hardware	Runtime software	Hardware/ Runtime software

TABLE 2-1. A comparison of several fast local processor memory alternatives. Conventional and Named-State register files define a *register name space* separate from the *virtual address space* used by the rest of the memory hierarchy.

Some of the properties, such as latency and bandwidth, are implementation dependent. It is relatively easy to build fast, multiported register files, but it is also possible to build fast, multiported caches. The method used by the hardware to select lines within each memory is also implementation dependent. Data caches typically use direct-mapped or set-associative lookup, while register files directly select lines in the file. The Named-State Register File described here uses a fully-associative lookup to associate names with register lines.

An important distinction between data caches and register files is that registers occupy a separate address space from the rest of the memory hierarchy. A data cache is addressed using a virtual or physical memory address, and may cache any program data. Conventional register files use a short index to select a register, while the NSF uses an expanded name space consisting of Context ID and offset.

Cache lines are typically allocated and managed using a fixed hardware algorithm. Some data caches allow the program to explicitly allocate cache lines [68]. This allocation is only useful in writing new data spaces, since the compiler must guarantee that the entire cache line will be written, to avoid incorrectly allocating partial lines [46].

A conventional register file allocates registers for sequential code blocks under compiler control [16]. In cases where the sequential call sequence is well known, the compiler can also manage registers across sequential procedure calls. For very dynamic or parallel programs, a conventional file must use runtime software routines to share the register file among many concurrent contexts [85].

The Named-State Register File may use the same compiler techniques to allocate registers for sequential code. A program could manage those registers in the same way as conventional register files for code where the activation order is known. But for dynamic and parallel programs, the NSF can dynamically manage registers in hardware, spilling and restoring variables as needed. This added flexibility can significantly improve the performance of many sequential and parallel programs.

Table 2-2 shows a similar comparison between a conventional register file, the Named-State Register File, register windows [31,66], and the C-machine stack cache [25,11].

Property	**Register File**	**Named-State File**	**Windowed Register File**	**CRISP Stack Cache**
Latency	< 1 cycle	< 1 cycle	< 1 cycle	1 cycle
Bandwidth	≥ 3 words/cycle	≥ 3 words/cycle	≥ 3 words/cycle	3 words/cycle
Selection	Direct select	Fully-associative	Direct select	Direct select
Contents	Register variables	Register variables	Register variables	Stack Frames
Name Space	Register name	Register name	Register name	Virtual Address
Addressing	Short index	CID + index	Window + index	Short index
Sequential Allocation	Compiled	Compiled/ Hardware	Compiled	Compiled
Sequential Management	Compiled preload	Hardware/ Compiled	Runtime preload	Runtime preload
Parallel Management	Runtime software	Hardware	Runtime software	Runtime software

TABLE 2-2. A comparison of different register file structures.

A windowed register file selects registers within the current active window, much like a segmented register file. It also manages register window frames for sequential programs with a combination of hardware support and runtime trap handlers [14]. Several researchers have proposed ways of multithreading between several concurrent threads using register windows [36,49,4]. All of these schemes use some form of runtime trap handler to switch between threads, and to swap threads into and out of the register file.

The C-machine stack cache is a variant of a traditional top of stack buffer. The C-machine pre-decodes instructions to turn stack references into offsets in the stack buffer. Since the stack buffer caches a region of virtual memory, locations in the stack are addressable as memory locations. A program may generate pointers to locations in the stack or the stack cache, and the processor must detect references to cached regions of the stack. The C-machine is slow to switch contexts between concurrent threads, since the entire stack cache must be flushed to memory, and a new stack must be reloaded for the new thread.

2.2.4 Advantages of the NSF

The Named-State Register File uses a combination of hardware and software to dynamically map a large register name space into a small, fast register file. In effect, it acts as a cache for the register name space. It has several advantages for running sequential and parallel applications:

- The NSF has low access latency, and high bandwidth.
- Instructions refer to registers in the NSF using short compiled register offsets, and may access several register operands in a single instruction.
- The NSF can use traditional compiler analysis [16] to allocate registers in sequential code, and to manage registers across code blocks [78, 86].
- The NSF expands the size of the register name space, without increasing the size of the register file.
- The register name space is separate from the virtual address space, and mapping between the two is under program control.
- The NSF uses an associative decoder, small register lines, and hardware support for register spill and reload to dynamically manage registers from many concurrent contexts.
- The NSF uses registers more effectively than conventional files, and requires less register traffic to support a large number of concurrent active contexts.

2.3 NSF System Issues

This section discusses some issues involved in managing the register name space and Context IDs, and the impact of data caches on NSF performance.

2.3.1 Context Identifiers

The NSF addresses registers using the concatenation of two short fields: a Context ID and register offset. The Context ID serves several purposes:

- Each Context ID uniquely identifies one of many concurrent activations.
  The processor can switch activations by switching to a new CID.
- Context IDs expand the size of the register name space.
  The NSF reserves a contiguous set of registers in that name space for each CID.

- The processor can manipulate registers associated with a CID as a unit.
  For instance, the *dealloc_CID* instruction deallocates all registers from a particular activation.
- CIDs decouple the register name space from the virtual address space.
  A programming model can enforce any mapping of CIDs to addresses by setting up entries in the Ctable.
- CIDs decouple the register name space from a potentially large context name space.
  Context IDs do not identify all contexts in existence on the processor, but only a small subset of those contexts that may have registers in the NSF. It is the responsibility of the program or the thread scheduler to allocate CIDs to procedure or thread activations. In this way, a choice of CID size in the NSF does not constrain the number of contexts that a program can spawn, nor the scheduling of those contexts.
- Context IDs and offsets are both short fields, which simplifies and speeds up the NSF decoder and Ctable hardware. A typical <CID:Offset> pair might be 10 bits wide.

The NSF provides a mechanism to handle multiple activations, but does not enforce any particular strategy. Since Context IDs are neither virtual addresses, nor global thread identifiers, they can be assigned to contexts in any way needed by the programming model. NSF design decisions should have no effect on the procedure and thread scheduling that the NSF can support.

The penalty for all this flexibility is that the set of Context IDs are yet one more limited resource that a program or thread scheduler must handle. The scheduler must map a potentially unbounded set of activations into a very few CIDs. The section that follows will argue that this should not be a serious problem for most practical programming models.

2.3.2 Managing Context IDs

The NSF does not constrain how Context IDs are assigned. While details of CID mapping and thread scheduling are beyond the scope of this thesis, this section suggests some examples of how programming models might use CIDs.

Sequential programs:

The compiler may allocate a new CID for each procedure invocation. Calling routines can create a register context for a callee routine, then pass arguments directly into the callee's registers. This is the model used by sequential programs in this study. Each called procedure is assigned a CID one greater than its caller. If the call chain is very deep, such that all CIDs are already in use, the program must reclaim some CIDs from the start of the chain. In order to reuse a CID, an activation must ensure that no registers belonging to that CID are still resident in the NSF.

A sequential program would normally spill registers from the NSF into stack-allocated, fixed-sized frames in memory. Each new procedure invocation simply increments the stack pointer, and writes that address into the Ctable entry for its CID. This is very similar to the way that register windows [31,67] are used on some RISC processors.

If all CIDs are in use, the run-time software must reuse a CID in order to allocate a new context. A CID that is no longer resident in the register file can be reused simply by updating its entry in the Ctable. But if a CID still has some lines resident in the register file, they must be explicitly copied out to memory and the CID deallocated from the register file before it can be reused. Even with the *probe* instruction of Section 2.1.5, this is still an expensive operation. Using a wide Context ID field can reduce the frequency at which context management software re-uses active CIDs. Certain software conventions can also reduce this burden.

However, typical sequential applications spend considerable time within a range of procedure nesting depth [35]. While call chains may become quite deep, the calling depth may not change at a rapid rate. This is the principle that allows register windows to capture a significant fraction of the procedure call frames without excessive window traps. A windowed register file with 10 register banks may overflow on less than 5% of calls for many simple applications [35]. By analogy, a NSF with a 4 bit wide CID field may only need to re-use a CID every 30 procedure calls. This is borne out by the simulations in Chapter 5, in which a 5 bit CID field holds most of the call chain of large sequential applications.

Fine-grained parallel programs:

A parallel language might allocate a new context for every thread activation. As discussed in Section 1.2, dynamic parallel programs may spawn many parallel threads. Many threads are able to run simultaneously, and a processor may rapidly switch among a small active set of threads in order to avoid idling. This is the model used by the parallel programs in this study [21].

Since threads in dynamic parallel models may be halted and then restarted, the thread scheduler must allocate room to store the thread's state in memory. These thread activations are usually heap allocated in a wide activation tree. In order to spill registers from the NSF file, the scheduler need only write the activation address for each new Context ID into the Ctable. This adds very little cost to the thread setup time.

Managing Context IDs for parallel code is more difficult than for sequential code, since an entire tree of contexts may be active simultaneously, rather than simply a sequential call chain. For most dynamic parallel programs, the scheduling of threads is data dependent and cannot be predicted at compile time. As in the sequential case, a thread scheduler may occasionally need to reuse Context IDs.

However, as noted in Section 1.2, any realistic thread scheduler must exploit locality in spawning threads and in running them. Since each thread consumes resources, the scheduler must often limit parallelism after a number of threads have been spawned [20,48]. In addition, efficient schedulers may exploit temporal locality by giving preference to a small active set of threads, before allowing any thread from outside that active set to be scheduled to run. This matches the model used in the NSF: a small number of contexts are resident and running in the register file, and a somewhat larger set of potentially active threads can be swapped in if all resident threads halt on synchronization points. A relatively small CID field should still allow an efficient thread scheduler to pick threads to run.

Large-grain parallel programs:

A programming model may support a mix of parallel *tasks* and sequential procedures. This may require a mixture of the two approaches discussed above. A large parallel task may run many procedures, and maintain its own stack. A processor could run that large task for some time, allocating contexts for each procedure activation. Then, if the task blocks, the processor could switch to another task, and run its procedures. This is the model used by Mul-T [52].

One way of allocating CIDs to large tasks is to devote a range of CIDs to each task, so that it can use that range of contexts for procedure calling. A trivial case is when a parallel program has been statically divided into a fixed number of large tasks. The compiler or linker may divide the CIDs on a single processor by the number of static tasks allocated to that processor. Thereafter, each task would behave much like a sequential program.

An alternative is to dynamically allocate CIDs to tasks and procedures, without any fixed division among the tasks. The choice of approach depends upon the number and average run length of tasks, and the frequency of task switches. The NSF itself does not limit the task scheduling options, but merely provides mechanisms to exploit them.

Other scheduling options:

The NSF is not limited to the programming models and scheduling strategies presented above. Many other strategies are possible. A programming model may allocate two Context IDs to a single procedure or thread activation. This would be useful if a compiler could not determine an execution order for basic blocks within that activation, and could not efficiently allocate registers among those basic blocks. (Such is the case for TAM [72] programs). Devoting two CIDs to the procedure would make register allocation easier, yet still allow basic blocks to pass data within the register file.

It would be inefficient to dedicate several frames of a segmented register file to a single procedure activation, and only use a few registers per frame. But in the NSF, since registers are only allocated as they are used by a program, there is relatively little cost to giving a small set of registers a unique CID. Only the active registers of that context will consume space in the NSF.

Task scheduling for parallel programs is an active area of research. Managing a set of Context IDs should not add much to the cost of thread scheduling and context allocation [55,85,61]. In many cases, CID handling should only require reading and writing entries in the NSF Ctable. The short time spent in this run-time software must be balanced against the cost of larger CID fields, which reduce the frequency of Context ID reuse.

2.3.3 Data Cache Performance

A processor's data cache affects the performance of the Named-State Register File in several ways. Because of cache interference effects, data caches miss more frequently while supporting multiple concurrent threads than a single instruction stream. In addition, the data cache must be able to quickly respond to register spills and reloads from the NSF, to prevent long pipeline stalls. This section addresses each of these issues in turn.

Several recent studies have investigated the effect of multithreading on cache miss rates and processor utilization. Agarwal [1] has shown that for data caches much larger than the total working set of all processes, the miss rate due to multithreading increases linearly with the number of processes being supported. Typical cache miss rates due to multithreading range from 1% to 3%, depending on the application. Weber and Gupta's experiments [29] confirm this behavior.

However, these experiments were for large, coarse grain applications. In some cases, to simulate higher levels of multithreading, the experimenters merely ran additional copies of the original sequential program trace, rather than dividing that program into smaller concurrent threads. Although there are no studies of the effect of multithreading for fine grain programming models, experiments have shown that cache miss rates are strongly dependent on a program's working set size. Since fine grain parallel programs have much smaller working sets than coarse grain programs [69], miss rates due to multithreading should not be a dominant component of execution time.

All multithreaded machines spill and reload registers to the cache. However, this small number of registers per context does not take much room in a data cache. The NSF caches fewer registers than segmented register files, since it only allocates spills registers for live data, and explicitly deallocates registers when possible. Finally, to minimize interference between concurrent contexts, [1] suggests hashing Context IDs to addresses, and accessing the data cache with the resulting hashed address. The NSF Ctable easily supports this technique.

An important distinction between the NSF and a conventional segmented register file is that when the latter switches contexts, it loads all of the registers from the context in one operation. While this loads unnecessary data, it also provides opportunities for pipelining those reloads. In addition, a segmented register file only reloads contexts after switching threads or calling subroutines. The NSF, on the other hand, may potentially miss and reload a register on any instruction. The effect that this has on processor performance depends on the design of the data cache.

Suppose that the data cache can respond to a fetch request that hits in the cache in time T_{hit}. Suppose that each successive access to that cache line takes time T_{pipe}. If the register write port is only one word wide, then the time required to reload a context of C words[1] into a segmented register file is:

$$Tseg_{reload} = T_{hit} + T_{pipe}(C-1) \qquad \text{(EQ 2-1)}$$

If after returning to a context, a Named-State Register File must reload N words in order to finish the current thread, this will take time:

$$Tnsf_{reload} = T_{hit} \cdot N \qquad \text{(EQ 2-2)}$$

For many modern RISC processors with on-chip data caches [57], T_{pipe} is equal to T_{hit}. But since cache access time determines the cycle time of many of these processors, T_{pipe} may be less than T_{hit} on future designs [13]. The Named-State Register File must then reload proportionally fewer registers in order to perform as well as the segmented register file. In other words, the NSF will perform better if:

$$\frac{N}{C} < \frac{T_{pipe}}{T_{hit}} \qquad \text{(EQ 2-3)}$$

The same arguments apply for register spilling, which could use write buffers into the cache to pipeline writes. Chapter 3 revisits this issue. But note that this analysis assumes that conditional register misses and reloads can be performed as efficiently as load instructions on a conventional processor. Given the level of pipelining in most modern processors, this may add additional complexity or latency to the implementation.

2.4 Design Alternatives

Line Size

Building a Named-State Register File requires a number of design decisions. The most basic issue is the width of register lines. As discussed earlier, a line may contain a single

1. Ignoring the effect of instruction fetching and dispatching.

register, or a short vector of registers. The line is the basic unit of granularity of the register file. While instructions may access individual registers within a line, an address is associated with each line in the register file.

An NSF register file could be organized with lines as large as an entire context. Such a register file would be very similar to the segmented register file described in Section 1.3. However, an NSF with large lines is still a fully-associative structure. Lines can be allocated and deallocated at any position in the register file. Any line in the file can hold any address, and an address may move from place in the NSF during the lifetime of a program.

A segmented register file, on the other hand, is a non-associative or indexed structure. Each context is assigned a frame in the register file, and a fixed address. Contexts do not automatically spill and reload to arbitrary frames in the file. Some segmented files support a fixed, round-robin frame allocation scheme in hardware [67]. While performs well for sequential programs, a round-robin frame allocation scheme performs poorly when interleaving multiple threads [36].

The line size of an NSF register file is a trade-off between area and granularity. One advantage of large lines is that fewer fully associative decoder cells are needed to address the register array. Another is that each decoder compares fewer address bits as line size increases. Thus, amortizing a decoder over several registers in a line can significantly reduce the size of the register file. However, as discussed in Section 1.3, large lines reduce the register file utilization, since only a few registers in each line will be active at any time. Reloading a large line may also load a number of inactive and unnecessary registers.

Multi-word lines and wide ports into the register file can improve register spill and reload times. As discussed in Section 2.3.3, each data cache access requires a significant initial latency. If the data path from the cache into the register file is wide enough, it can amortize the cost of that access over several words. This may be enough to justify building Named-State Register Files with wide lines.

Register reloads

As mentioned Section 2.1.4, there are several possible strategies for handling register reloads in the NSF. The most efficient reload policy depends on the register line size, the complexity of NSF logic, and the performance of the memory system. Chapter 5 investigates the relative performance of different reload policies.

For misses on register reads, the NSF may reload the entire line containing the missing register (*block reload*), or simply allocate a line and reload the one missing register (*sub-block reload*). In order to reload individual registers into a line, each register in the line must be tagged with a *valid* bit. The valid bit indicates that the register contains live data. Every register reload or write must set the valid bit of its target. A NSF could then allocate a line in the register file on a read miss, and only reload one register, setting its valid bit.

The NSF could avoid reloading the remainder of the line until a read miss on another register in that line.

An NSF could assign a valid bit to each line in the file, each register, or some sub-block size in between the two. As discussed above, the choice of sub-block size may depend in part on the width of the path to the data cache, and the initial latency and pipelined performance of the data cache. If reloading two adjacent registers requires no more time than reloading one, it may be best to reload a two word sub-block on every miss.

As shown in Section 5.4.3, valid bits and sub-block reloads may dramatically reduce register reload traffic. Valid bits can also reduce register spill traffic, since only valid registers need be spilled when a line is replaced. However, as shown in Section 3.5, valid bit logic requires a significant fraction of the register file area. Reloading an entire line on misses is simpler to implement, and does not require any additional logic inside the register file itself.

Writes may also miss in the register file. In a similar manner, a write miss may cause a line or sub-block to be reloaded (a *fetch-on-write* policy), or may simply allocate a line and write a single register (*write-allocate* policy) [46]. Write-allocate has the advantage that the compiler can reuse one or two registers in a line without paying the cost of reloading the entire line if it has been swapped out. It also means that only register reads cause lines to be reloaded. On the other hand, it complicates the NSF spill and reload logic. A write-allocate policy normally requires valid bits on each register in the NSF.

A fetch-on-write policy requires merging the newly written register with the rest of the reloaded line or sub-block. Fetch-on-write can be very inefficient, especially on the first write to a new line. A fetch-on-write policy forces the NSF to reload an empty line on the first write to the line.

One way to improve the performance of fetch-on-write without adding valid bits to the NSF is to have the compiler explicitly tag each write to a new line. This informs the NSF that it should just allocate a new line for that register, rather than missing and trying to fetch the line from memory. As shown in Section 5.4.2, tagging writes can eliminate a large fraction of write misses and reloads. Tagging to allocate lines is easy if lines are the size of an entire context, but much more difficult if lines are a few registers wide [46].

However, if lines in the NSF are only a single register wide, there is no need to ever reload a line on a write miss. Each write simply allocates a new line and writes the register. All writes to such a register file are explicity write-allocate. As shown in Section 3.6, single register lines also simplify register allocation logic.

Explicitly deallocating a register after it is no longer needed frees up space in the register file. This can increase the utilization of the register file, ensuring that it only contains live data. The NSF provides a tag line for *read-deallocate*, to read a register and immediately deallocate it. As in the case of write tagging, identifying the last use of a register in a subroutine requires effort by the register allocator. A compiler may need to generate two versions of a code-block: one for execution within a loop, and one for the final iteration to deallocate registers that are no longer needed. In case the compiler is not able to explicitly deallocate each register, the NSF also provides a signal to deallocate all registers belonging to a particular Context ID in one cycle.

Register spilling

The NSF spills lines to memory when it becomes full of live data. One way of managing the register file is to allow the file to fill completely, and then stall on an instruction that allocates a new line. A simpler method, from a hardware standpoint, is to always maintain a free line in the register file. This ensures that register writes always complete, regardless of which register is being written. The NSF can then stall the following instruction, if necessary, to spill a line from the file. The NSF prototype chip described in Section 3.7 used the latter method. However, for simplicity, the software simulations in this thesis followed the former strategy.

Context ID

Another design decision is the width of the Context ID field used to address registers in the NSF. As mentioned earlier, larger CID fields allow more contexts to be active simultaneously, without reusing CIDs. Large CIDs increase the size of the fully-associative register address decoders. As shown in Section 3.5, depending on the organization, the address decoders may consume 8% to 15% of the NSF chip area. For multi-ported register files, decoder area is not proportional to the number of CID bits, since several ports may share the same CID.

Larger CID fields also increase the size of the Ctable which translates Context IDs to Context Addresses. However, this single ported lookup table is relatively inexpensive to build, and is not in the critical path for register accesses.

Registers per context

There have been several studies of the working set of sequential programs, and of the optimum size of register windows for RISC processors [26,12]. This research indicates that is often more efficient to divide a large register set into windows of 12 or 16 registers each, rather than half as many windows of 32 registers each. There is little improvement in program performance as additional registers are added to each window.

The Named-State Register File, on the other hand, can support many registers per context without requiring each context to use those registers. In effect, each procedure activation, or each thread, uses only as many registers as it needs. No activation will spill live data out of the register file without replacing it with more recent data. This may contradict those earlier studies, and argue for allowing large register sets per context in the NSF.

However, even in the NSF, there is a cost to allowing each context to address many registers. Larger register offset fields increase the size of instruction words, since each instruction must be able to directly address three operands. In the same way, larger offsets increase the size of the NSF address decoder.

This thesis will not fully investigate the optimum number of registers per context for different applications. To effectively study this trade-off would require modifying the register allocation phase of a compiler, and generating code optimized for different maximum context sizes. Those compiler modifications for both sequential and parallel code are outside the scope of this thesis.

Additional register ports

Additional ports into the register file have proved useful for super-scalar processors, which must feed several functional units simultaneously. They could also be useful to improve the performance of the Named-State Register File. An additional read port allows the NSF to spill lines from the file as it fills, without interrupting the instruction stream. This port does not require additional complexity in the associative decoder, since the NSF decides internally which victim lines to spill.

Additional write ports to the NSF are not as useful, since the NSF does not support any form of register pre-loading. At best, increasing the width of the existing write port allows the NSF to reload several registers at a time from the data cache.

2.5 Analysis[1]

Intuitively, a fine-grain binding of names to locations in the Named-State Register File would seem to require fewer loads and stores than a segmented register file to support the same number of active threads. The simple analysis in this section bears this out.

Consider a processor with N words of fast register storage[2]. Assume that the number of live registers per task is an exponentially distributed random variable with mean μ. To support i tasks on a conventional multithreaded processor, the registers are divided into i frames of N/i words each. With this arrangement a spill occurs in a single frame with a probability of $e^{-N/i\mu}$. So the probability of a spill in any frame is given by:

1. This analysis was first presented by Prof. William Dally.

2. This analysis does not consider name spilling which occurs when the number of names that can be bound to fast register storage is exhausted. Nor does it consider the impact of different replacement strategies in the NSF.

$$P(MTspill) = 1 - (1 - e^{-N/(i\mu)})^{i} \qquad \text{(EQ 2-4)}$$

The probability density function for the total number of live registers is obtained by convolving the density functions of the tasks together giving an Erlang distribution:

$$f_{liveregs}(x) = \frac{(x/\mu)^{i-1}}{\mu(i-1)!}e^{-x/\mu} \qquad \text{(EQ 2-5)}$$

In a Named-State Register File a spill occurs only when the total number of live registers exceeds *N*. The probability of such a spill is given by integrating Equation 2-5 from *N* to ∞ giving:

$$P(CCspill) = e^{-N/\mu}\sum_{r=0}^{i-1}\frac{(N/\mu)^{i-1-r}}{(i-1-r)!} \qquad \text{(EQ 2-6)}$$

Figure 2-4 compares the spill probabilities of a segmented register file and a Named-State Register File as the number of tasks, *i*, is varied from one to 32 with the number of registers, *N*, fixed at 128, and the average number of live registers per task, μ, fixed at 8.

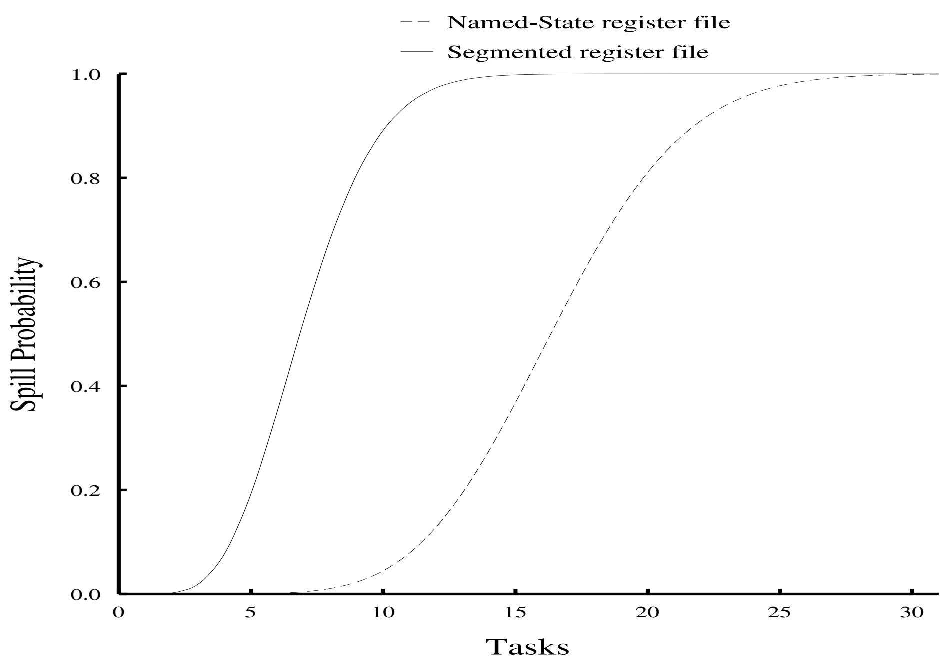


FIGURE 2-4. Probability of register spilling for a segmented register file and the NSF. Number of registers: N=128, Average live registers per task: μ=8.

Figure 2-4 shows that the NSF has a significantly lower probability of spills than a conventional segmented register file. With eight active tasks, half of the register storage holds live variables. At this 50% utilization level the NSF has only a 1% probability of spilling while the segmented register file has a 69% spill probability. Alternatively, the NSF could attain this spill probability while accommodating eighteen active tasks, many more than a segmented register file.

CHAPTER 3
Implementation

The preceding two chapters introduced the Named-State register file, described how it is accessed, and how it supports multiple contexts efficiently. But in order to be useful, the NSF must also provide fast access to data and must be feasible to build in existing VLSI technology. This chapter describes how to build a NSF, compares its size and speed to ordinary register files, and demonstrates a prototype implementation.

Preface

Register files are *multi-ported*, meaning that they read and write several operands simultaneously. A typical processor reads two register operands and writes a single destination register per instruction cycle. The register file could be built with fewer ports by time-multiplexing within a processor cycle. This is how the MIPS-X register file [18], for example, can support two reads and a write per cycle with a two-ported register file. Other high performance processors require additional read and write ports to support multiple functional units or speculative execution [24]. However, most of this discussion will describe single cycle, three ported register files.

3.1 NSF Components

Figure 3-1 describes the detailed structure of the Named-State register file. It also highlights some differences between the NSF and conventional register files. The primary difference is the NSF's fully-associative address decoder. Before discussing the details of register decoders in Section 3.2, this section reviews the other major components of the NSF.

3.1.1 Inputs and Outputs

Table 3-1 lists the signals that connect the Named-State register file to a processor pipeline and memory system. The *Address* buses tell the register file which registers to read and write, while *Data* buses carry those operands to and from the ALU. Register addresses for the NSF require more bits, but otherwise these buses are the same as in conventional register files. However, the NSF also uses four *Alloc* signals in addition to the address bits to indicate when to allocate and deallocate registers.

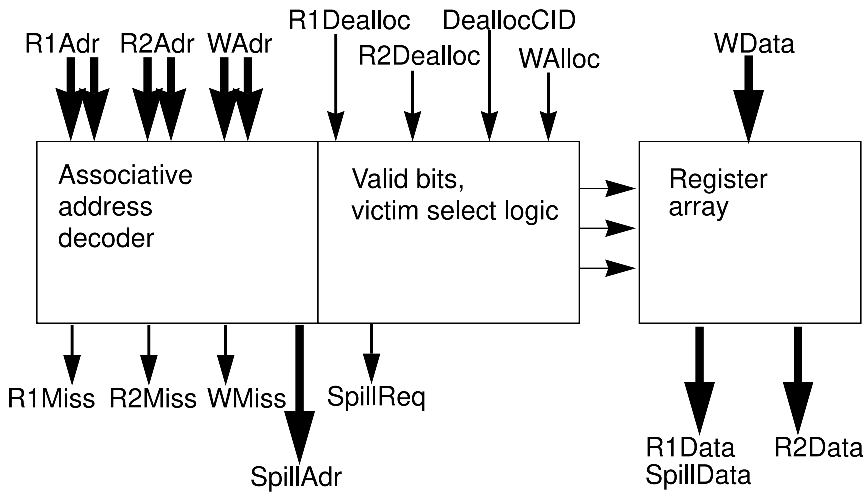


FIGURE 3-1. Input and output signals to the Named-State Register File..

Register File Signal	**Type**		**Definition**
R1Adr, R2Adr, WAdr	Address	Input	Address of register read operands 1, 2, and write operand. Each address has the form: <Context ID: Offset>
WData	Data	Input	Data value to write into register.
R1Data, R2Data	Data	Output	Register read operands.
R1Dealloc, R2Dealloc	Alloc	Input	Deallocate a register after reading its value.
WAlloc	Alloc	Input	Allocate this write operand, do not fetch from memory.
CIDDealloc	Alloc	Input	Deallocate all registers from the context specified in R1Adr.
R1Miss, R2Miss, WMiss	Status	Output	An operand was not resident in the register file. Stall pipeline while fetching a line from memory.
SpillReq	Spill	Output	The register file is full. Spill a line to memory.
SpillAdr	Spill	Output	The register address of a spilled line.
SpillData	Spill	Output	Contents of spilled line, output through a read port.

TABLE 3-1. Signals linking the Named-State register file to processor pipeline.

The NSF also uses three *Status* signals to indicate whether an operand was resident in the register file. In response to an access miss, the pipeline will usually stall while fetching the appropriate register line from the data cache. After reloading the line into the register file, the instruction can proceed. The `probe` instruction, introduced in Section 2.1.5, is an exception to this schedule. Probe instructions simply test whether an operand is in the register file, but never reload or spill lines. Also note that the *WMiss* signal is not needed for files that are only one register wide.

Finally, the NSF uses *Spill* signals to unload data from the register file when it becomes full. The *SpillReq* signal stalls the processor in order to spill a line. Just as the NSF reloads data through the register write port, it spills lines through one or both of the register read ports. The *SpillData* bus simply shares those ports with the pipeline. However, the NSF uses one additional port (*SpillAdr*) to spill a line's Context ID and Offset at the same time as its data. This register address is translated to a virtual memory address by a table lookup as described in Section 2.3.2. This allows the NSF to spill registers into the data cache.

3.1.2 Valid bits

Each register in the NSF must be tagged with a *valid* bit. This indicates that the register has been allocated in the file, and contains a valid data word. If a line of the file contains several registers, each must have a valid bit. When a line is first *allocated* for a register, only that register's valid bit will be set. (This is known in cache terminology as *sub-blocking).* Subsequent writes to other registers in the line set their valid bits as well. As described in Section 2.1.3, a *read_deallocate* operation will clear the valid bit associated with a register after reading the value of that register. Finally, the NSF must be able to deallocate all registers belonging to a single context by clearing all their valid bits in one operation.

3.1.3 Victim select logic

The victim select logic determines when the Named-State register file becomes full, and also picks which line to spill to make room for new registers. To simplify the pipeline, the NSF should always maintain one free line in the register file. This ensures that register writes always succeed, and allows instructions to drain from the pipeline. If an instruction writes the last free line in the register file, succeeding instructions must stall to allow the NSF to spill a line. Full detection logic simply checks that all register lines have a valid bit set.

The NSF uses a *Pseudo Least Recently Used* strategy (PLRU) [37] to pick which line to spill from the file. It approximates a true LRU strategy using a single *access* bit per line. The NSF sets this bit on any read or write to a line. A rotating *victim pointer* points to the first line whose access bit is not set. When all access bits are set, they are all cleared, and the victim pointer is advanced by one line. This ensures that the NSF never flushes registers that have recently been accessed. This strategy takes advantage of the temporal locality of register accesses among several different contexts.

The NSF could use other victim selection strategies, such as *round-robin*, which steps through each line in sequence, moving to a new line every time the register file fills up.

This is simple to implement, but completely ignores the order in which registers were accessed. A simple variant of the *round-robin* strategy never flushes a line that has just been written, to avoid thrashing in the register file. Finally, a *random* or *pseudo-random* strategy simply picks a victim at random from among the register lines.

3.2 Fixed Register File Decoders

The most significant difference between the Named-State register file and conventional register files is the NSF's fully associative address decoder. Before describing how to build such a programmable decoder, this section reviews how to build address decoders for conventional register files.

Conventional register files use *fixed* address decoders. The register file is divided into N rows, each M registers wide. An address R bits wide (where $R=log_2N$), selects one of the N rows. If each row contains several registers, other address bits select one of the registers to read or write.

Figure 3-2 shows a single row of such a register file. Each of three decoders in each row checks if an operand address matches the index of this row. If so, the decoder drives the appropriate *word line* to read or write this row of the register array[1].

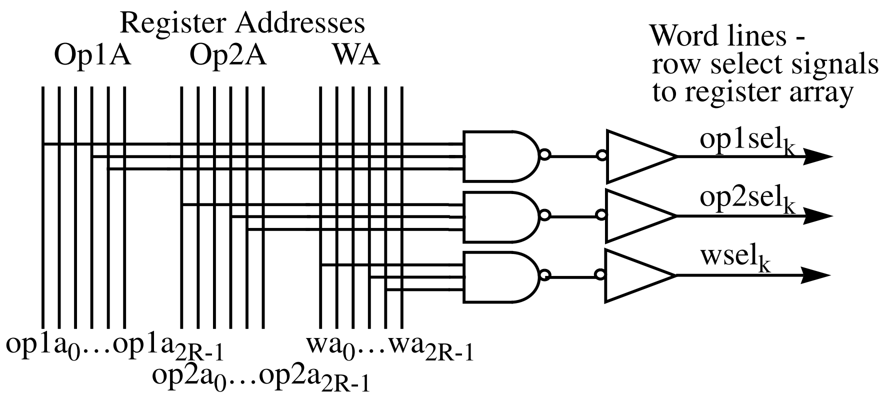


FIGURE 3-2. A row of a conventional register decoder.

There are many ways to decode a register address. The one-stage decoder, shown in Figure 3-2, checks to see if the address bits encode its row number by performing an R bit

1. Logic external to the register file ensures that it will never simultaneously read and write a single register within the array. Instead, the logic enables *bypass* paths to directly carry the output of one instruction to the input of the next.

wide AND. Buffers drive R bits of an operand address and R bits of its inverse down across each row decoder.

An alternative is to use a tree of decoders for each row, in which each AND gate is only B bits wide, rather than R bits. Each stage of the tree is faster than a single stage decoder, since gate switching speed is a function of fan-in. This structure also allows successive levels of the decoder to buffer up the signal in order to drive a large capacitance word line.

A final alternative is to *pre-decode* the register address bits, and use a two-level decoder structure. If the R bit operand address is composed of M fields of B bits each, the first decoder stage converts each field of B bits into 2^B decoded signals, and drives these signals down across each row. Each row decoder AND gate is then only M bits wide.

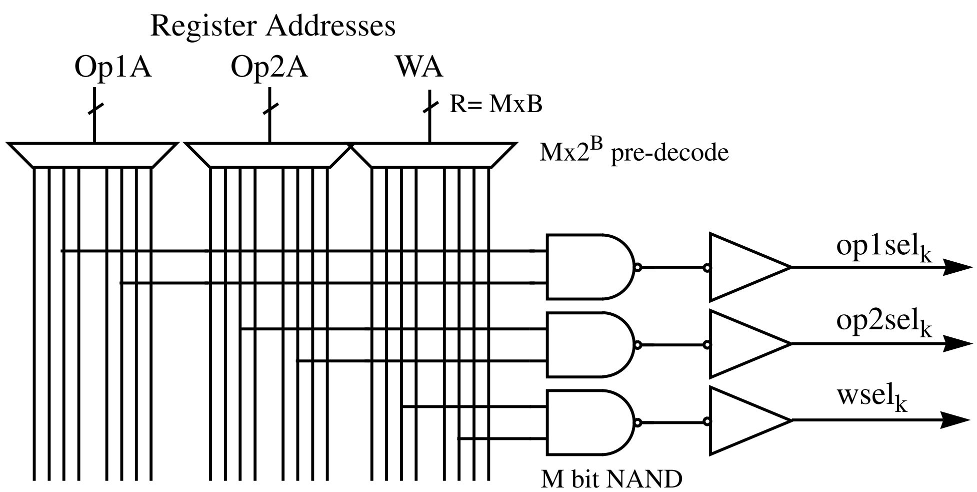


FIGURE 3-3. A two-stage, fixed address decoder.
Each of M pre-decoded fields is 2^B bits wide , in which only one line is active.

Since the first stage of decoding is combined with address drivers, this structure may be as fast as a one-level decoder, with much lower fan-in per gate. This design can also be built with the same number of buffered address lines as in the one-level decoder. If B=2, the first stage converts each field of two address bits into four decoded signals, for a total of $2R$ signals. Because of its speed and compactness, a pre-decoded structure is often used to address large register files.

3.2.1 Decoder circuit design

Although the decoders shown above perform a NAND across several address bits, that NAND need not be a simple combinational gate. A combinational M input NAND gate requires $2M$ transistors in CMOS technology, and does not scale well with the number of inputs.

Decoder gates can also be built as precharged NAND structures. These designs pull up on an output line during one clock phase, and then pull down on that output through a chain of pass transistors during the next phase. While precharging requires more complicated clocking strategies, and is only suitable for synchronous circuitry, it only requires $2+M$ transistors. A precharged design can be faster than a fully combinational design for small M.

In an alternative precharged NOR gate, each input can discharge the output through a single transistor. While this design adds more capacitive load to the output signal, it scales better than NAND structures for large M.

3.3 Programmable Decoders

The Named-State register file uses a programmable address decoder to build a fully associative structure. While each row of a fixed decoder is wired to recognize its index on the address lines, the NSF decoder is first programmed with an address, and then matches against that address on subsequent accesses. Figure 3-4 describes the behavior of such a

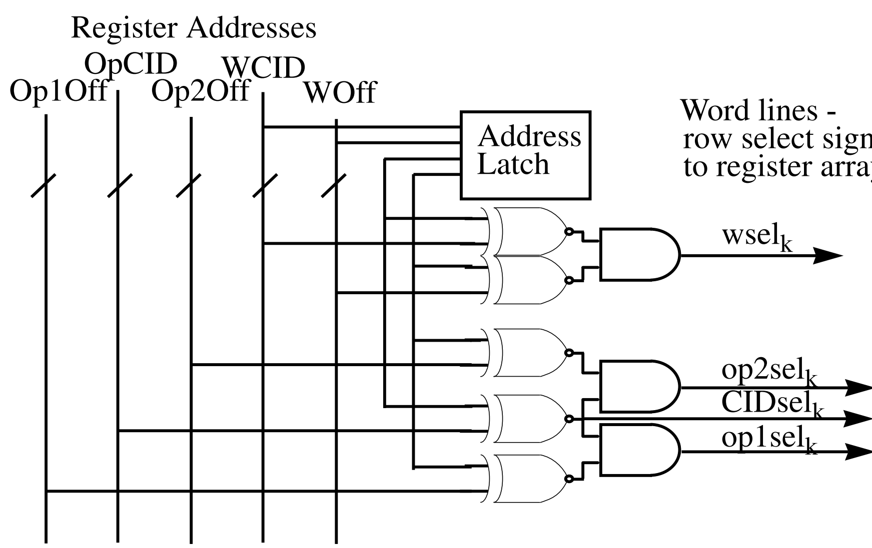


FIGURE 3-4. A row of a programmable address decoder.

programmable decoder. Each row of the decoder contains a latch that holds the address of this register line. An address is written into the latch when this row is allocated to a register variable. Each of three operand decoders performs a multi-bit comparison of the latched address with an operand address.

Since a register address in the NSF consists of a *Context ID* and an *Offset*, each row decoder must latch that full address. However, as shown in Figure 3-4, the two read operands share a common CID, while the write operand uses a different CID. This allows the NSF to copy registers from one context to another, and for successive instructions to come from different contexts. But since no instruction needs to read from different contexts at the same time, the NSF can economize on address lines.

In the structure shown here, each row matches against CID and offset separately. This allows an instruction to deallocate all rows that belong to a particular context. It also has the advantage that each XNOR gate has fewer inputs. A two-level structure can be as fast as a single level for this number of address bits.

3.3.1 A programmable decoder cell

While Figure 3-4 describes a row of a decoder as separate latch and match logic, it is smaller and faster to build as an array of *Content Addressable Memory* cells [33]. A CAM cell is a latch that matches against its address lines. Each row of the NSF decoder consists of *C* CAM cells to match against Context ID bits, and *O* cells to match Offset bits.

Figure 3-5 is a schematic diagram of an address match unit for Offset bits, showing one bit of a simple CAM cell. This cell contains a one bit address latch, as well as three sets of

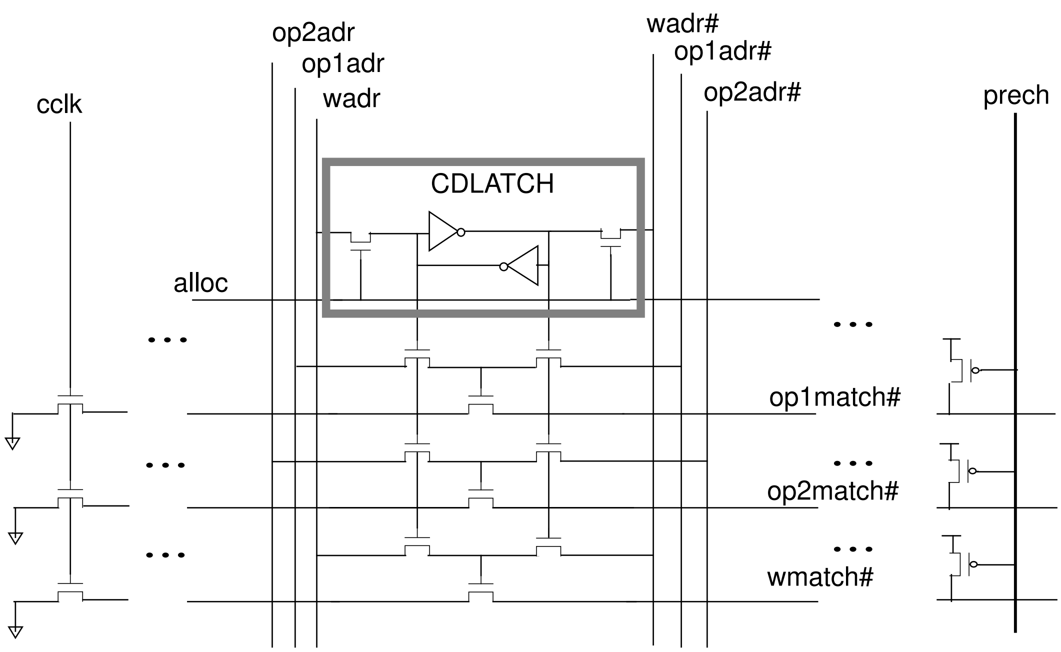


FIGURE 3-5. A single bit of a programmable address decoder.

operand address match logic. Initially this row of the register file is empty. When the NSF allocates this line, it asserts *alloc* for this row. This loads a new address into each latch bit *CDLatch* in the row. On subsequent accesses, if this bit of any of the operands matches the value in *CDLatch*, the match logic enables a bit of the precharged NAND gate. If all the bits match, the decoder asserts a word line into the register array to read or write that register.

The bit cell shown in Figure 3-5 matches one of three operand offsets against the address offset of this row. A reduced form of this bit cell compares two Context IDs with the CID of this row. A final set of NOR gates combines the two match results and drives the word lines. Each NAND chain is short enough that the decoder is not significantly slower than conventional fixed decoders.

3.3.2 Alternative decoder designs

Figure 3-5 describes a very simple decoder circuit design. Other designs may be faster or more efficient. The thesis does not investigate those circuit designs, since the goal is to evaluate the effect of different architectural decisions on register file performance. This section discusses some alternatives.

The precharged NAND circuit is as fast as a NOR gate for a small number of address bits. Since CID and offset should each be 5-6 bits wide, there is no advantage to using a NOR structure, which requires additional signal lines to clock properly. Alternatively, a programmable decoder could use pre-decoded address lines, using a NOR across 2^B decoded bits and then a NAND of M results. However, this would require $Mx2^B$ bits of address latch, rather than MxB bits for a single-level decode.

This design uses a single transistor clocked with *cclk* to discharge the entire NAND chain of match transistors. An alternative for both NAND and NOR designs would be to gate operand addresses with a clock. If both *op1adr* and *op1adr#* are held low until the discharge phase, then the final pull-down transistor to ground on *op1match#* is not needed. This might speed up decode, and eliminate some signal lines, but would significantly complicate clocking strategy for the circuit.

Another technique for speeding up the decoders is to attach a sense amplifier to each match line. The sense amplifier could detect when the line begins to discharge, and drive the word line much faster. Otherwise, a simple NOR gate must wait for the match lines to completely discharge through a long chain of pass transistors.

By way of example, a new design for a memory management unit exploits some of these techniques to build a 64 entry translation look-aside buffer [33]. The fully associative address decoder matched across 22 address bits in a precharged NOR structure with sense amps. The TLB had an access time of 4.35ns in a foundry ASIC process.

3.4 Performance Comparison

The *Spice* circuit simulator [58] was used to evaluate the performance of the Named-State register file as compared to conventional register files. Several different designs were tested for each file. This section describes the results of those simulations.

The goal of this study was not to build the very fastest register file decoders. Instead, the simulations compared two decoders built in a conservative process, with very simple logic. No complex clocking schemes or fancy circuit techniques were used in this comparison. Instead, both decoders were designed using the same level of effort, in hopes that the relative performance would scale to more aggressive designs.

Each simulation traced the time required to decode a register address and drive a word line into the register array, and to read out the contents of a register. The simulations did not count time required to drive signals to and from the processor pipeline, or additional multiplexers on the data path. The circuits were synchronous with a global clock *cclk*, which allowed them to use precharged decoding logic. The simulations assumed that register addresses were stable and could be driven across the decoders before the start of the decode phase. All decoder logic was allowed to settle before gating with *cclk*.

These simulations used a relatively conservative, *1.2* μ*m* CMOS process. It is a two-level metal process, with an average capacitance of *0.3 fF/*μ*m* for minimum width first level metal. The simulations used the size of an SRAM cell in this process to estimate the height and width of different register file organizations. The simulations estimated the total capacitance on signal lines by adding the gate or drain capacitance of transistor loads on the lines to the wire capacitance.

The decoder for the "segmented" register file used pre-decoded address lines and a pre-charged NAND pull-down decoder for each operand. Two organizations were tested: a decoder for a 128 row by 32 bit wide register file, as well as a 64 row by 64 bit word design. The latter design required each decoder to match across fewer address lines, but added additional load to the row word line drivers.

The same two register file organizations were simulated with programmable, "Named-State" address decoders. These designs used pre-charged, NAND pull-down circuits to decode the bits. The associative decoder did not pre-decode any address bits, since each row decoder must match across all bits of the address anyway. Pre-decoding does not save area for programmable decoders. For the number of address bits required, NAND pull-down structures were somewhat more compact than pre-charged NOR circuits, and were just as fast.

Figure 3-6 shows the results of these Spice simulations. The NSF required 12% to 23% longer to decode addresses than a pre-decoded, segmented register file, since it must

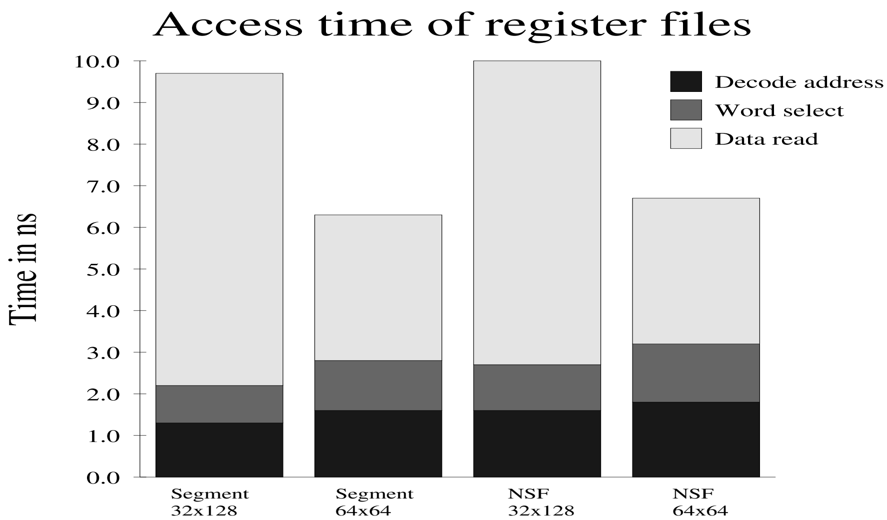


FIGURE 3-6. Access times of segmented and Named-State register files. Files are organized as 128 lines of 32 bits each, and 64 lines of 64 bits each. Each file was simulated by Spice in *1.2*µ*m* CMOS process.

compare more address bits. It also took 15% more time to combine Context ID and Offset address match signals and drive a word line into the register array.

For both register file sizes, the time required to access the Named-State register file was only 5% or 6% greater than for a conventional register file. This may affect the processor's cycle time, if the register file is in the critical path. Currently, many processors are instead limited by cache access time [34].

3.5 Area comparison

This section compares the VLSI chip area required to build a Named-State register file with that required for a conventional register file. The area estimates were derived using cell layouts from a prototype NSF implementation described in Section 3.7. That chip was built using an inefficient *2*µ*m* N-well CMOS process, with two levels of metal interconnect.

Figure 3-7 illustrates the relative area of the different register files. In *2*µ*m* technology, a 128 row by 32 bit wide Named-State register file with one write and two read ports is 78% larger than the equivalent segmented register file. An NSF that holds 64 rows of two registers each requires 46% more area than a segmented register file.

Figure 3-8 shows the same area comparison, if the register files were built using a more aggressive *1.2*µ*m* CMOS process. Because the NSF programmable decoder can be laid

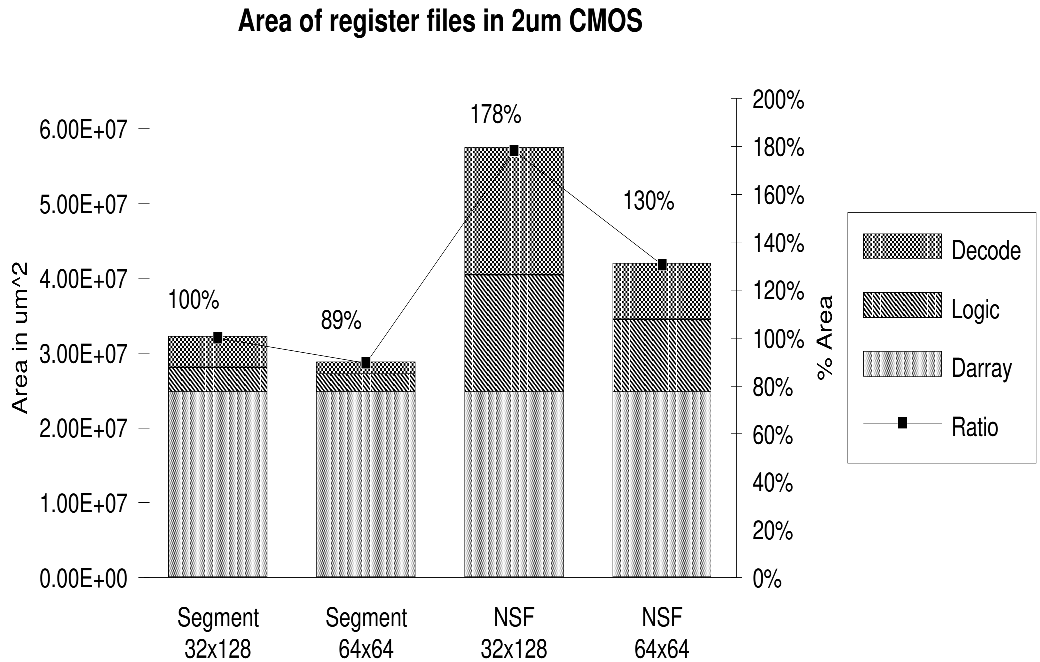


FIGURE 3-7. Relative area of segmented and Named-State register files in 2um CMOS. Area is shown for register file decoder, word line and valid bit logic, and data array. File sizes are expressed as bits wide x registers high. All register files have one write and two read ports.

out more compactly under the address lines, and because metal pitch shrunk more than minimum feature size, the decoder consumes a smaller fraction of the register file area in this technology. However, the random logic for valid bits and victim selection is a larger fraction of chip area. In this technology, a 128 row by 32 bit wide Named-State register file is 54% larger than the equivalent segmented register file. An NSF that holds 64 rows of two registers each requires 30% more area than the equivalent segmented register file.

3.6 Design Alternatives

Line Width

Figure 3-8 shows the effect of line width on register file area. A NSF file with two registers per line is 20% smaller than a file with single register lines. This comes from amortizing decoders and valid bit logic across more bits of registers. A file with wide lines also matches fewer address bits in the programmable decoder. A two register wide file is only slightly slower than a file with one register per line.

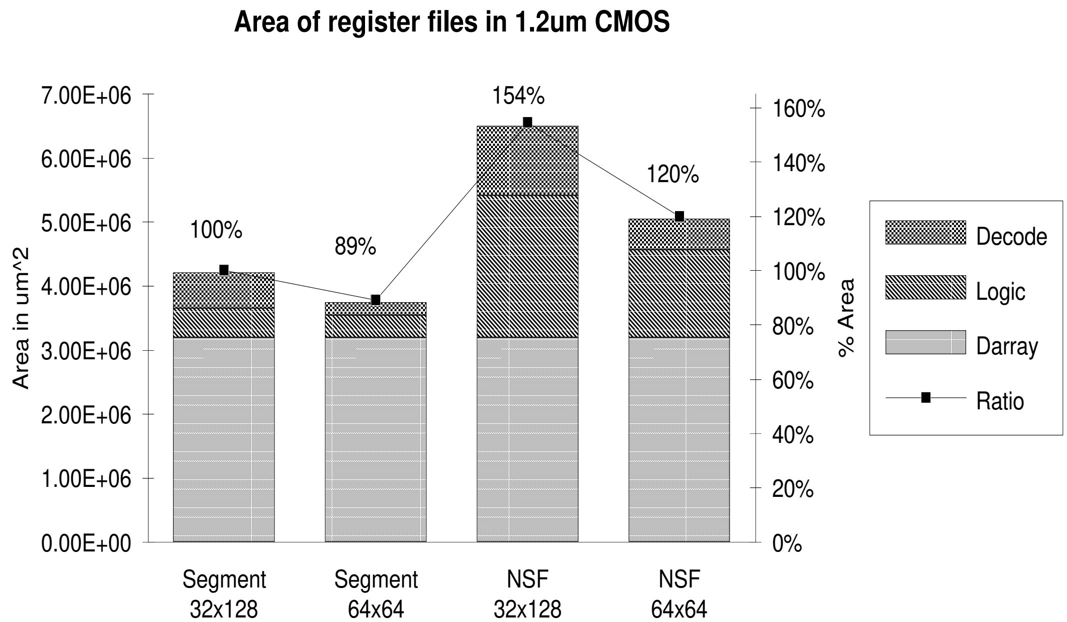


FIGURE 3-8. Relative area of segmented and Named-State register files in 1.2um CMOS. Area is shown for register file decoder, word line and valid bit logic, and data array. All register files have one write and two read ports.

Register spilling

Another alternative concerns how to flush registers. When the NSF spills a register, it must spill both data and address. One way of reading out the address bits is through the write operand bit lines *wadr* and *wadr#*. Another is to widen the register array, and store address bits alongside register data. A final alternative is to provide additional ports out of the register file, so that a register can be spilled while others are being read and written. All the simulations and area estimates in this thesis use the first alternative.

Additional ports

Many recent processors require more than two read ports and one write port into the register file. Extra ports may be used to service additional functional units, to support speculative execution, or to allow spilling and reloading of registers in the background. Figure 3-9 estimates the relative area of segmented register files and the NSF, each with two write ports and four read ports. Note that the NSF files in this comparison can read from one context, and write to two other contexts in a single cycle. (The CID decoders are three ported).

As ports are added to the register file, the area of an NSF decreases relative to segmented register files. In this comparison, a 128 row by 32 bit wide Named-State register file is

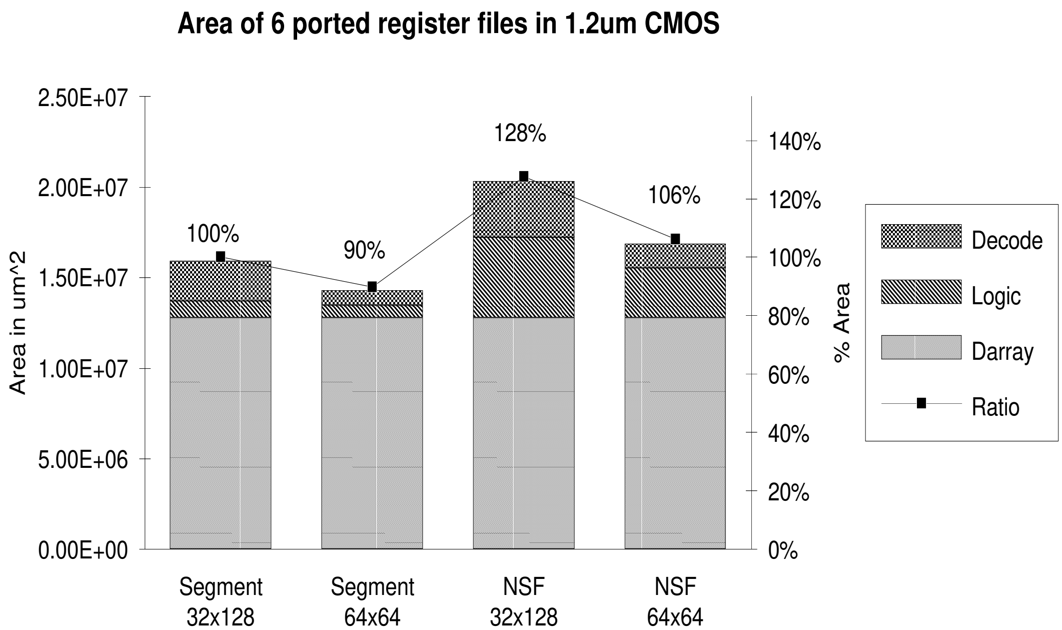


FIGURE 3-9. Area of 6 ported segmented and Named-State register files in 1.2um CMOS. Area is shown for register file decoder, word line and valid bit logic, and data array. These register files have two write and four read ports.

only 28% larger than the equivalent segmented register file. A 64 by 64 bit wide NSF is only 16% larger than the equivalent segmented register file. In these estimates, the area of a multiported register cell increases as the square of the number of ports. Decoder width increases in proportion to the number of ports, while miss and spill logic remains constant.

3.7 A Prototype Chip

A prototype chip was built to evaluate the Named-State register file. The goals in building this chip were:

- To design and evaluate the NSF logic in detail.
- To show that the NSF could be built in conventional VLSI technology.
- To investigate the performance of the NSF, and to verify circuit simulations.
- To validate area estimates of different NSF implementations.

The chip logic was designed by Peter Nuth and David Harris. The chip was designed and laid out by David Harris over the summer of 1992. The chip was fabricated by the MOSIS fast prototyping service. Figure 3-10 shows a photograph of the completed chip.

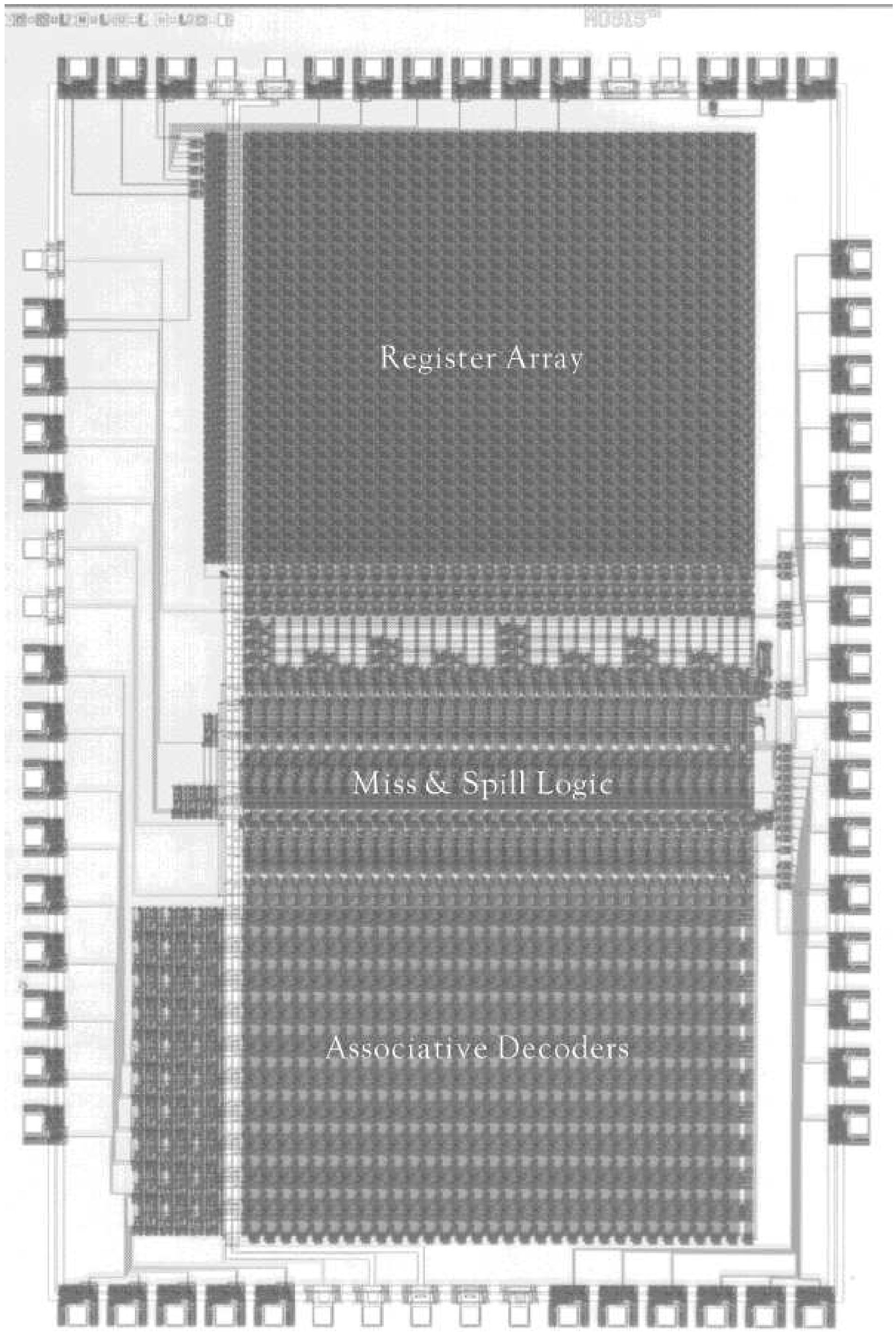


FIGURE 3-10. A prototype Named-State Register File.
This prototype chip includes a 32 bit by 32 line register array, a 10 bit wide fully-associative decoder, and logic to handle misses, spills and reloads. The register file has two read ports and a single write port.

3.7.1 Design Decisions

- The chip is fabricated in the MOSIS double metal, 2μ*m* N-channel CMOS process. While this is not a dense process, it was the only one that our CAD tools could support at the time.
- The chip holds a 32 line by 32 bit register array, with one register per line. This was the smallest realistic register file that we could build. Using single register lines simplified decoder and valid bit logic.
- The prototype chip used 10 bit wide register addresses, 5 bits of CID and 5 bits of Offset. This is clearly too large for such a small register array, but it allowed us to evaluate the performance of larger register files. A 10 bit address would be appropriate for a 128 register NSF.
- The address decoder was built as described by Figure 3-5, using two 5 bit wide precharged NAND chains. The prototype does not use any unusual circuit techniques.
- The prototype could read two registers and write a third on every cycle. Unlike NSF designs described here, the prototype chip could be addressed with three distinct CIDs simultaneously. Since most realistic processors would share a single CID between two read operands, the prototype address decoder is larger than necessary.
- Rather than a pseudo-LRU victim selection strategy, the prototype chip used a round-robin scheme to pick a register to spill. A simple rotating shift register selected the next victim.
- The chip used a conservative, static logic design for valid and miss logic.
- Finally, since the prototype used single register lines, every write operation attempted to allocate and write a new register in the array. A write never missed, since the chip logic always maintained at least one free line in the array. On every register write, the NSF wrote a data word into that free line. If the write address did not match any existing line in the file, the NSF set a valid bit on the free line, allocating that register. Otherwise, if the write address already existed in the file, the data word was written to that existing line.

Appendix A describes the structure and operation of the prototype chip in more detail. That report also describes the purpose and logic design of each of the basic cells in the prototype.

CHAPTER 4
Experimental Method

4.1 Overview

This chapter outlines the software simulation strategy used to evaluate the Named State Register File. The results of those simulations are detailed in Chapter 5.

Figure 4-1 describes the simulation environment. At its core is a flexible register file simulator called *CSIM*. The simulator is driven by traces of parallel and sequential benchmarks. The filters *S2NSP* and *TLTRANS* translate the assembly code of those benchmarks to NSF instructions. The filters also annotate the assembly code so that when it is run on a conventional architecture, it generates a full trace of the program's execution. CSIM reads both translated NSF code and program traces, and simulates accesses to a register file of arbitrary size and organization.

4.2 Register File Simulator

Figure 4-2 outlines the modules of CSIM, the register file simulator. The core of the simulator is a flexible register module, and a pipeline control unit that issues reads and writes to the registers. Other modules fetch instructions, interleave streams, and handle code and trace files.

4.2.1 Simulator Instructions

In order to simulate application programs, we generate traces of *basic blocks*, not full instruction traces. A basic block is a contiguous sequence of instructions between branch points. The code translators S2NSP and TLTRANS must be able to parse native assembly code and recognize branch instructions and branch targets. In return for this extra complexity, block traces are much smaller and faster to generate than instruction traces. Typical basic block traces for the applications used in this study are 80MB - 200MB long. A full instruction trace would be approximately 20 times longer. More importantly, this strategy allows CSIM to read in the entire program as NSP code before beginning the simulation. By paging in the block trace during execution, CSIM requires far less disk bandwidth than an instruction trace approach. As a final optimization, the block tracing routines compress traces on the fly to further reduce their size by a factor of eight.

CSIM executes a very simple instruction set known as *Icode*. Icode abstracts away all details of the instructions executed. In fact, Icode does not specify ALU operations at all,

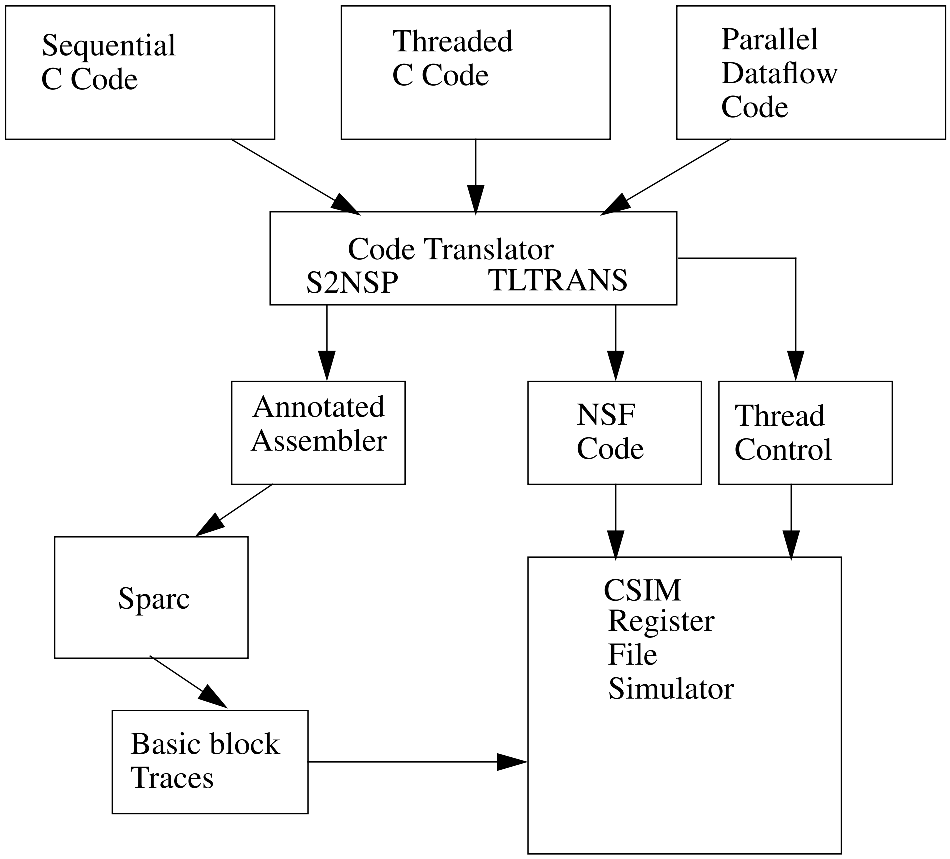


FIGURE 4-1. Simulation environment.

merely distinguishing register to register instructions from loads and stores or branches instructions. Table 4-1 describes the Icode instruction types.

The only unusual Icode instructions are the context control instructions `NEWCID`, `POPCID`, and `SWITCHCID`. An ordinary subroutine call does not automatically create a new context for the child procedure register frame. Rather it is the called procedure's responsibility to allocate and deallocate a context as needed. `NEWCID` creates a new context identifier, but does not explicitly allocate any registers for the new context. Subsequent instructions from this thread will use this new context identifier to allocate, read and write registers. The `POPCID` instruction, on the other hand, explicitly flushes any of the

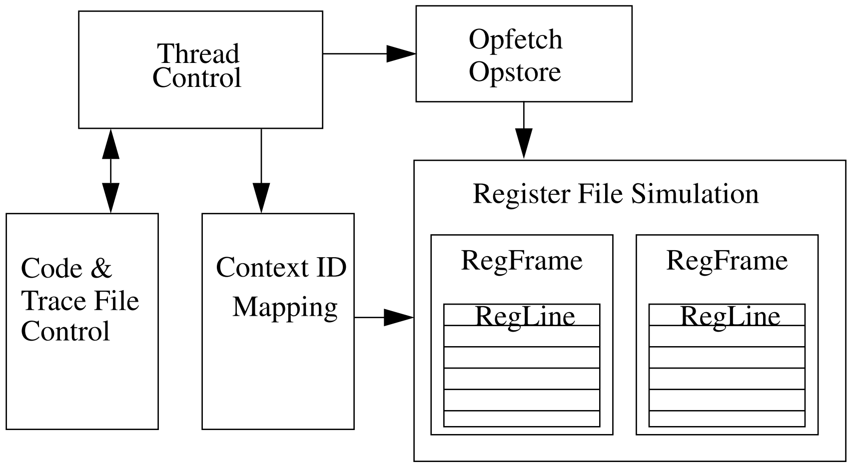


FIGURE 4-2. The CSIM register file simulator.

Instruction	Description
NOP	Do nothing
REG	Any register to register instruction
LOAD	Load word from memory into register file
STORE	Store register into memory
BRANCH	Jump to a new basic block
CALL	Subroutine call to a new basic block
RETURN	Return from subroutine
NEWCID	Create new context
POPCID	Deallocate this context from register file
SWITCHCID	Switch to an existing context

TABLE 4-1. Icode instruction types.

current context's registers that remain in the register file. The `SWITCHCID` instruction switches to a new context as determined by thread control routines. (See Section 4.2.2).

Each Icode instruction may specify several source register operands and a destination register. CSIM permits the register accesses described in Table 4-2.

Register access	Operation
READ	Normal register read. Miss if register is not resident.
READ_DEALLOC	Read register, then deallocate.

TABLE 4-2. Register access operations.

Register access	Operation
WRITE	Write register. Miss if register is not resident.
WRITE_ALLOC	Allocate register line, then write.

TABLE 4-2. Register access operations.

A normal `READ` access simply reads the value of a register. If the register is not resident in the register file, the pipeline stalls and must fetch the register from memory. `READ_-DEALLOC` explicitly deallocates the register when it is no longer needed. If all the registers in a line have been deallocated, the entire line is cleared from the register file.

`WRITE_ALLOC` allocates a new register line and writes a register within that line. A normal `WRITE` operation assumes that the register addressed is resident in the register file. If it is not, the simulator must fetch that register line from memory. This distinction allows a compiler to label first writes to a register line, so that the instruction will never cause a write miss on an unallocated location. However, if the register file is organized with only a single register per line, there is never any reason to miss on writes. The simulations take this into account, executing all `WRITE` operations as `WRITE_ALLOC` for single word register lines.

As mentioned in Section 2.1.3, each read or write may access a register in the current context, or in its parent context. Thus a single instruction may copy a value from a caller's register frame to the callee's frame. Each access can also set or clear a register full bit for synchronization on a data word.

4.2.2 Thread control

The CSIM simulator can run three types of programs:

- Sequential code.
- Coarse grain parallel code formed by interleaving several sequential programs.
- Pure parallel programs.

CSIM can issue register accesses from a single instruction stream or from several concurrent streams. To emulate multithreaded code, CSIM can interleave instructions from a number of distinct sequential programs. By switching threads in response to run-time events, CSIM can simulate the behavior of parallel programs. Otherwise, an external parallel processor simulator may generate traces of a true parallel program running on a single processing node. These traces specify both basic block and context, to represent any number of thread scheduling policies. If such a parallel simulation environment is available, it relieves the burden of specifying a realistic scheduler within CSIM.

The benchmarks described in the chapters that follow are all either purely sequential programs traced on a conventional processor, or parallel programs traced by a parallel processor simulator. When running a purely sequential program, CSIM allocates a new context to each new procedure invocation. This allows register contexts to be used as stack frames, in a manner similar to register windows [66]. Instructions may copy data from a parent's context to its child for argument passing, or from child to parent to return the result of a procedure call.

When interleaving multiple sequential threads, CSIM creates a new context for each procedure invocation by each of the threads. Instructions may pass data between parent and child procedures, or between concurrently running threads[1]. CSIM may switch between threads in response to a number of run-time events, such as a procedure call or return, or a register operand miss. CSIM can simulate an external data cache in order to switch threads on a cache miss, or may simply force a cache miss on some fraction of cache accesses. CSIM tracks full/empty bits on register locations in order to set and clear synchronization points on those registers. However, no provision is made to automatically restart a thread when a synchronization variable has been resolved. Finally, CSIM may emulate a HEP style [76] multithreaded pipeline by running the threads in a round-robin manner, issuing a single instruction from each thread before switching to the next.

In order to run a pure parallel program on CSIM, the program must have generated a trace file that indicates both which basic blocks to run and when to switch contexts. Otherwise, CSIM would have to fully emulate the thread scheduling policy, as well as the semantics of that parallel program language. The parallel program traces used in this study consist of basic block and context identifier pairs. The TAM parallel programming language [21] used in this study ensures that synchronization points always occur at the end of basic blocks, so that each block runs to completion, and context switches occur only at basic block boundaries.

4.2.3 Context ID management

In order to support the three different thread execution models described in Figure 4-1, CSIM tags each instruction executed with a unique *context number*. The context number is a three word tuple: $\{\text{Task}\,|\text{Iteration}|\text{Call\_Depth}\}$. The *task* field distinguishes statically defined task traces. This allows CSIM to run multiple copies of the same trace, each rooted at a different context number. A task may fork any number of child threads at run-time. Each child is assigned a new *iteration* number within the current task. Finally, a sequential or parallel thread may call and return from subroutines. Specific NSF instructions in the code trace increment the context number's *call_depth* field after a procedure call to allocate a new context for each call frame.

1. Note that this requires a thread to know the context number of a particular invocation of another thread. This is not well supported by existing sequential languages.

A given instruction stream, labelled with a unique $\{\text{Task} \mid \text{Iteration}\}$ pair, has a single call stack. As the stream calls and returns from procedures, it increments and decrements the *call_depth.* Since each stream has only one call stack, the tuple always uniquely identifies an existing context.

While a *context number* can address an unlimited number of contexts, the number of contexts that may be resident in the register file is quite small. So the NSF uses a short *context identifier*, or *CID*, to address contexts in the register file. The field used to represent the CID need only be large enough to refer to all the contexts in used by concurrently running threads. Depending on the size of the register file, a typical CID field might be 5 or 6 bits wide, large enough to refer to 32 or 64 concurrent contexts.

The *context map* module of CSIM translates context numbers associated with threads to CIDs for addressing the register file. It also translates a CID to the address of a context in memory when flushing a register out of the file. This is the role of a *Ctable*, as described by Section 2.3.2. Every time the simulator switches to a new thread, it checks that the CID assigned to that thread is still valid. Since there are a very large number of potential context numbers competing for a relatively few CIDs, the context map may have to reassign CIDs among the currently running threads. In practice this happens infrequently, due to the locality of threads scheduled for execution.

4.2.4 Opfetch and Opstore

Ordinarily, CSIM will fetch and execute successive instructions from a single thread, until hitting an exception as described above. Every NSF instruction may read two source registers and write a third register. CSIM decodes each instruction's source registers, builds a register address, and fetches each in turn from the register file. If any source register is not available in the register file, the instruction has *missed* and must be re-issued when the operands become available[1]. At this point, the CSIM pipeline may elect to switch contexts and fetch instructions from a new stream, or it may simply stall the current instruction while fetching the missing register line. All the simulations described in Chapter 5 simply stall the pipeline long enough to fetch a line from the data cache.

4.2.5 Register file

The central module of the CSIM simulator is the model of the register file itself. This module allocates and deallocates registers, tracks all accesses, and reports hits and misses to the opfetch and opstore units.

1. Note that currently, CSIM fetches the registers in sequence. The same instruction may miss and be reissued several times if neither source register is available. Real hardware would fetch both missing registers before re-issuing the instruction.

The register file is organized in a hierarchical structure. At the lowest level is the register cell. For every cell in the register file, CSIM tracks whether the cell contains valid data, whether the data is more recent than data in memory, and whether the `FULL` bit is set for synchronization on that register value.

Register cells are organized into lines. A register line is the unit of granularity of the register file. Lines are flushed out and reloaded into the register file as a unit. CSIM keeps track of whether a line is resident in the register file. It also tracks the number of live registers in each register line. When a line is flushed out of the register file, CSIM does not modify any register cell contents except the dirty bits. In this way, when the line is reloaded into the register file, valid registers are restored correctly.

A register frame contains all the register lines associated with a particular context. Unlike cells and lines, register frames are not physical structures. They exist only to make the simulation more efficient. At any time, any or none of the lines belonging to a particular context or frame may be resident in the register file. The `NEWCID` instruction creates a frame for a new context, but does not load any of its lines into the register file. The `POPCID` instruction deallocates all the frame's resident lines, and destroys the frame.

The top level register file structure in CSIM handles register accesses, flushes and reloads register lines, and counts a large number of statistics during a simulation.

The register file can hold a fixed number of register lines. Each line could belong to any existing register frame. A number of different policies are possible for selecting which register line to flush when the register file is full. A specific module of the simulator handles this victim selection. Currently, CSIM uses a *Least Recently Used* strategy, in which the victim is the register line that has not been read or written for the longest time. The register file miss handler keeps pointers to all register lines, sorted by the order in which they were accessed. The line at the tail of the list is the victim to be flushed. CSIM could support other victim selection strategies by using the same structures as the current miss handler. A *Round-Robin* policy orders the lines by their position in the register file. A *Random* policy picks a victim at random from the resident register lines. All of the simulations in Chapter 5 use the LRU strategy.

Before each CSIM simulation run, a user must specify the parameters of the register file to be simulated. These include:

- The number of registers in the register file. In this way, simulations compare equal sized register files that are organized differently.
- The number of registers in each register frame. Different programming languages and applications may access a different number of registers per context. All simulations set the size of the register frame to the maximum register offset addressed by any context.

- The number of register cells in each register line. This is the granularity of binding of variable names to registers. A register line may range in size from a single register to the size of a register frame. The former simulates a fully-associative Named-State register file, while the latter simulates a segmented register file.
- The maximum number of contexts that may be addressed concurrently. This is limited by the size of the Context ID field used to address the register file.

4.3 Sequential Simulations

This section describes the method used to translate sequential programs to run under CSIM, and some of the implications of that translation process. CSIM requires two input files: the translated Icode for an application program, and the trace of basic blocks executed by a run of that program. The code translator *S2NSP* performs both functions for a sequential program.

4.3.1 S2NSP

The S2NSP translator converts Sparc [79] assembly code to Icode. It also inserts instructions into the Sparc code that trace the basic blocks executed by the program. While S2NSP could translate any program compiled for the Sparc, we have only used C [50] language source programs for sequential benchmarks.

With some minor modifications, S2NSP could translate the assembly code of any conventional RISC processor to Icode. The reason for choosing Sparc code is that the Sparc architecture uses register windows [67]. Register windows, as fixed sized register frames, are similar to contexts in the Named-State register file. In addition, since Sparc programs pass procedure arguments in registers, they are a good test for passing values between contexts. Finally, Sparc code can be converted to acceptable quality Icode with a relatively simple translator.

To convert a Sparc program to Icode, S2NSP allocates a new context for each new register window required. In a manner similar to the Sparc, it allocates the context after entering the first code block of a subroutine. Only procedures within the user's program will allocate contexts, but not operating system routines that cannot be translated by S2NSP. However, an operating system routine may call a procedure in the user program. Allocating contexts at the time of the call would lead to incorrect execution.

S2NSP translates Sparc instructions to the simple Icode instructions described in Table 4-1. It also translates references to Sparc registers to the corresponding registers in a CSIM context. The global, local and output registers of the Sparc register window become registers within the current context. A window's input registers become accesses to output registers in the parent of the current context.

While a Sparc instruction can address any one of 32 registers, the translated program will address at most 20 registers per context under CSIM. Only the 16 local and output registers belong to the current register window. Also, S2NSP reserves two Sparc global registers to help trace the program execution. Some other global registers are reserved for use by the operating system. The remainder can be used as scratch registers by any procedure, and are translated to registers in the current context.

4.3.2 Limitations

The Icode produced by S2NSP is not as efficient as that produced by a compiler specifically for the Named-State register file. The most serious omission is that code produced by S2NSP does not explicitly allocate and deallocate registers. A true Icode compiler would deallocate registers after their last use within a procedure. However, S2NSP would have to be far more complex in order to perform this register usage analysis. The translator would have to record register usage within basic blocks and within the entire procedure to see when it was safe to deallocate a register. In addition, S2NSP would have to generate prologue and epilogue code blocks to allocate or deallocate registers used in a code block within a loop. But even this analysis would not be enough to generate efficient Icode, since a compiler would usually assign registers differently to accommodate CSIM.

S2NSP does not reassign registers, but merely translates Sparc register references to the equivalent NSF registers. Since the NSF can support a large number of registers per context without the cost of saving and restoring large register frames, a true NSF compiler would probably use many more registers than the 20 used by the Sparc compiler. In fact, an NSF compiler might never reuse registers within a procedure, if it were always possible to deallocate the registers after their last usage.

Compiler construction and register assignment for the NSF is an interesting research topic in its own right. As always in computer architecture, there is a tension between hardware and software techniques to solve a particular problem. More sophisticated software could simplify the design of the NSF. Unfortunately, the scope of this thesis does not allow for such investigations.

Rather than building a more sophisticated compiler to generate Icode, the CSIM simulator was instrumented to see the effect of more efficient software. CSIM counts all occasions when a line is flushed out of the register file, and then never reloaded before the end of the context. If the line contains live data, it could have been deallocated after the last read of those registers, freeing up space in the register file. CSIM counts all such unnecessary flushes that any reasonable compiler would have prevented.

CSIM also counts all writes to new or previously allocated registers. Some writes miss on a register, and cause an empty line to be loaded into the register file. Had the compiler tagged this `WRITE` as a `WRITE_ALLOC` instead, it could have prevented this unnecessary register reload.

The statistics gathered by CSIM count obvious situations where deallocating on register reads and allocating on writes would improve performance. But since deallocating registers in one context could greatly reduce the number of loads and stores required to run another context, CSIM cannot determine the true performance of tagged reads and writes without actually simulating good code.

4.3.3 Tracing sequential programs

In order to trace an actual run of a program, S2NSP must insert tracing instructions into the Sparc assembly code stream. Then all the assembly language files are compiled and linked with additional tracing routines to generate an executable.

After identifying a basic block in the assembly code, S2NSP inserts a sequence of 14 instructions in-line at the beginning of the basic block. The instructions insert the unique index of this basic block into a buffer in memory. Just before every procedure return, or before the program exits, S2NSP inserts a call to a logging routine which writes out the block trace buffer. The logging routine prints the traces as ASCII digits in order to make debugging easier. However, it also pipes the output through a Lempel-Ziv [44] compression routine to reduce the size of the traces and the amount of disk I/O required.

The tracing code inserted in-line increases the execution time of the program by 20%. However, printing out the traces, which calls a standard C library printout routine to write out each trace index, increases the program's execution time by a factor of 20. The final phase, of piping the traces through a compression routine, only adds an additional 20% to the execution time.

4.3.4 Anomalies

The Sparc instruction set contains a number of anomalies that make it difficult to translate to Icode.

- The Sparc processor uses condition codes for all testing and branching. Since an arbitrary number of instructions may separate setting a condition code from the branch that depends on it, any instructions inserted into the code stream must not modify any condition codes. In particular, block tracing code cannot test an index into the trace buffer to determine if it should flush out the traces.
- Most branch instructions on the Sparc are delayed branches. S2NSP must re-order instructions around delayed branches when generating Icode. This is especially difficult in the case of "branch or squash" instructions, in which the instruction in the branch delay slot is not executed if the branch is not taken.
- The CSIM register file supports two register reads and a single write per cycle. Most Sparc instructions are simple, 3 operand instructions that execute in a single cycle. However, a store instruction may read three registers: the data to be stored, an address

in memory, and an offset from that address. In addition, the Sparc instruction set includes instructions to load and store double words. A double word load specifies a target register for the load. Both that register and the register that follows it in the register window are written in successive cycles. S2NSP must translate these instructions into multiple instruction Icode sequences.

- Sparc call and return instructions implicitly read and write the stack pointer register, one of the input registers in the register window. The Sparc instructions to push and pop register windows may read an arbitrary register in the parent window and write another register in the child window.
- Finally, the Sparc architecture uses a separate set of registers for floating point operations. CSIM does not model these registers at all. For this reason, none of the applications chosen for this study use much floating point arithmetic.

4.4 Parallel Simulations

Converting parallel programs to run under CSIM is similar to converting sequential programs. In both cases, the source program is translated to Icode, and a run of the application generates a basic block trace. However, there are not many true parallel programming environments that could be converted to run under CSIM. Some programming environments consist of a message-passing library called from within ordinary C programs [73]. Other approaches statically partition the application into a small number of very large threads that are each run on a separate processor [74]. Neither of these programming environments could generate code well-suited for execution on a multi-threaded processor. To evaluate the context switch performance of the NSF required an application with many short threads, and frequent switching between threads. For our parallel experiments, we used Dataflow programs written in Id [60], as compiled using the Berkeley TAM compiler [72].

4.4.1 TAM code

A goal of the TAM (*Threaded Abstract Machine*) project is to evaluate the performance of Dataflow programs on a number of different machine architectures. The traditional approach to running Dataflow code has been to build processors customized for that language [7,65]. Many of these machines treat each Dataflow instruction as an independent task, and synchronize between instructions. The Berkeley TAM compiler instead produces code for general purpose parallel computers [64,81]. The TAM compiler groups instructions into short sequential code blocks. The TAM run-time scheduler attempts to group together a set of code blocks to run as one unit. This technique reduces the number of context switches required to execute the Dataflow program, and produces code that runs on conventional sequential processors [79].

In order to produce code for a number of different processors, the TAM compiler compiles Id programs to a common intermediate language known as *TL0* [84]. TL0 represents the assembly language of a Threaded Abstract Machine, much like Pascal's p-code. A number of translators have been written from TL0 to different instruction set architectures. A portable translator converts TL0 to C code [50] and then compiles the resulting program for a specific machine.

Every TL0 subroutine is partitioned into a number of short *threads*[1]. Each thread executes to completion without suspending or branching. It is similar to the basic blocks of sequential programs. A thread may *fork* another thread by pushing a pointer to that thread onto a local thread queue known as a *continuation vector*. Threads on the thread queue are executed in LIFO order, so if *ThreadA* forks *ThreadB* and then terminates, it ensures that the processor will simply branch to *ThreadB*.

The only conditional statements in TL0 are conditional forks, which fork one of two threads onto the local queue. Threads may also be *synchronizing*, meaning that they are controlled by an *entry count* variable. Each fork of a synchronizing thread decrements and tests its entry count. The thread will not execute until its entry count reaches zero. This is useful to ensure that all variables used in the thread are available before running the thread. The compiler is responsible for allocating and initializing all entry count variables for an invocation of a subroutine.

Every TL0 subroutine also contains a number of *inlets*, or message handler routines. Inlets are similar to threads, in that they are short blocks of code that run to completion. However, since inlets respond to messages sent by other processors, they run asynchronously, and may interrupt other threads. An inlet can read and write variables within the current context, and may also fork threads. Inlets are used to receive arguments to a function, the results returned by child function calls, and responses to global memory requests.

Since TL0 is an explicitly parallel language, it assumes that any number of invocations of a subroutine may be running at one time. Each subroutine invocation is known as an *activation.* An *activation frame* is the context of a single activation. It contains all local variables for the activation, some local queues for scheduling threads within this activation, and a pointer to the code for this subroutine. Every thread forked by the activation is pushed onto its local thread queue. The TAM run-time system will run all threads on the local queue before deactivating the current activation frame and switching to a new activation. This ensures that the processor will run for as long as possible within a single context before switching to a new context. TAM will never schedule an activation for execution unless it contains threads that are ready to run.

1. Note that TAM "threads" are short code blocks, not the general parallel activations described in the rest of this thesis.

Since the threads of an activation are run sequentially, the only way of spawning parallelism in the TAM system is by sending a message. Sending a message to *Inlet0* of a subroutine allocates and initializes a new activation frame for that subroutine. Inlet0 allocates all local variables, sets up all entry counts for synchronizing threads, and initializes the local task queue. Similarly, *Thread0* of a subroutine terminates the current activation. It sends the result of the subroutine call in a message to its parent activation, and deallocates the current activation frame.

4.4.2 TLTRANS

The TLTRANS filter translates TL0 code to Icode for execution under CSIM. Every TL0 inlet or thread becomes an Icode basic block. Every TL0 instruction is translated into one or more Icode instructions. References to TL0 local variables and synchronization counters become references to CSIM registers.

Each TL0 synchronizing thread is translated into two Icode blocks: The first simply decrements the entry count for this thread and exits. The second basic block decrements the entry count and then runs the thread to completion. Since CSIM does not know the initial value of each synchronization counter, it relies on the trace of the TAM program to indicate each decrement and test of the counter, and the final successful execution of the thread.

A TAM activation frame is equivalent to a CSIM context. Unlike conventional stack frames, CSIM contexts can be saved to memory and resumed in any order. TLTRANS allocates a register in the context for each TAM local variable or synchronization counter. The size of the context required for each subroutine depends on the number of local variables in the subroutine. In order to simulate the resulting Icode, we must set the context size in CSIM to the size of the largest subroutine in the application.

4.4.3 Tracing TAM programs

While TL0 code is inherently parallel, it can also run on a sequential processor. A compiler known as *TLC* translates the TL0 code of an application to C code. All thread scheduling and message sending is handled by calls to a TAM run-time library. The resulting compiled C program is a simulation of a TAM machine running the application.

TLC can also produce annotated C code, which traces every thread and inlet as the program runs. It will also trace every synchronization counter, and every failed attempt to run a synchronizing thread. We have written a program to translate this verbose tracing output to a block trace for CSIM. The resulting trace specifies every basic block that was executed, and the context in which it ran. Running this trace under CSIM describes one execution of a parallel program on a sequential processor. Unfortunately, since thread scheduling decisions are compiled in with the application program, TLC does not allow a user to vary that execution at all.

4.4.4 Limitations

Register allocation

There is no register allocator yet available for the Threaded Abstract Machine. Register allocation in TAM is difficult, since thread may execute in any order within a quanta. While this allows threads to be grouped together and minimizes context swtiches, it makes variable lifetime analysis very difficult.

TAM sets no hard limit on the number of local variables per activation. As a result, there is a greater variance in the size of contexts in TL0 programs than in Sparc programs. Some TL0 subroutines will use only a few local variables, but TLTRANS will create a large, fixed size context for each routine. This means that many TAM programs will run very inefficiently with some register file configurations under CSIM. In particular, when CSIM is used to simulate a segmented register file, every context switch will require loading and storing very large contexts, even though the compiler may know that most of those registers were never used by a particular subroutine.

There are several ways of overcoming this problem with our translation system. The first would be to set a maximum context size in TLTRANS, and for the translator to only map some local variables to those registers, leaving the rest in the context's memory segment. However, since TLTRANS cannot do register usage analysis on TL0 code, it will not be able to pick the best subset of local variables to assign to registers.

The approach used in our simulations is to set a limit on the maximum context size in TLTRANS, and to simply map all local variable references into this small number of registers. So even if the application uses hundreds of local variables per thread, the simulator only touches 32 or 64 registers per context. This mapping is completely simplistic, and does not take into account variable usage frequency, or locality of references.

Thread scheduling

The preceding section highlights another problem with this method of tracing TAM code. We trace an execution of a parallel program on a sequential processor. All messages that the processor receives were generated by its own threads. In the absence of these messages, threads would all run on this processor, in depth-first order, just like a sequential program. Since inlets are asynchronous, they will interrupt this execution order. While this is not the most interesting run of that program, it is not clear whether a true parallel execution would show different register miss and reload rates under CSIM.

When a TL0 procedure calls another subroutine, it sends a message to that routine's *Inlet0*. The inlet handler creates the child's context, and runs within that context. Subsequent message sends pass arguments to the child routine through other inlets. A sequential processor, after running each inlet, will return to caller's context, and continue to run threads in that context. The processor will not switch to run any threads in the child context until all ready threads in the parent have completed.

This thread schedule is exactly that of a sequential program in which the caller creates a stack frame for the child, and initializes local variables in that frame. The child does not actually run until the parent is forced to wait for the result of the procedure call. Of course, a TL0 procedure may spawn several child routines concurrently, and may pass arguments to each, before invoking them in turn.

Since TAM software will compile and run on parallel processors, we could try to generate traces on different parallel machines to see if the code performs any differently than on a sequential processor. We could trace the threads and inlets executed on one node of a multi-computer and run that trace through CSIM. Unfortunately, the TAM software library does not readily generate thread traces on parallel processors. Modifying the TAM software or instrumenting other parallel simulators to produce these traces is outside the scope of this thesis.

CHAPTER 5
Experimental Results

This chapter describes the performance of the Named-State Register File running a number of sequential and parallel applications. The first section describes the benchmark programs used in this study. Later sections review the advantages of the NSF relative to conventional register files, and present simulation results that support these claims. Succeeding sections analyze the behavior of several different register file organizations, to see which factors contribute most to performance. The chapter closes by computing the overall effect of Named-State on processor performance.

5.1 Benchmark Programs

The performance results in this chapter are simulations of three sequential and six parallel programs. As described in Chapter 4, the sequential applications were written in C [50], compiled to Sparc assembly code [79], and translated to assembly code for the *CSIM* simulator. The sequential benchmarks are:

- ZipFile A public domain compression utility, based on the *Lempel-Ziv* [44] algorithm. Zip is compressing a single 8,017 byte file.
- RTLSim An register transfer language simulator for the *Message Driven Processor* [22]. RTLSim is running a 198 cycle network diagnostic on a single MDP node.
- GateSim A gate level simulation of the MDP. This program is a superset of RTLSim, using the RTL simulator to trigger a much more intensive simulation of the MDP gates. GateSim is running a 735 cycle suite of network diagnostics on the gate level model of the MDP router.

The parallel programs were written in Id [60], compiled and traced using the Berkeley TAM simulator [21], and translated to CSIM code. The parallel benchmarks are:

- AS A simple array selection sort.
- DTW Dynamic time warp algorithm.
- Gamteb Monte Carlo photon transport in a cylinder of carbon.
- QS Quick-sort using accumulation lists.
- Paraffins Generates paraffin molecules from radicals.
- Wavefront A multi-wave matrix manipulation algorithm.

As described in Section 4.3.2, the sequential programs were only able to use a maximum of 20 registers per procedure call. However, each parallel program defined many local variables per context, which were then mapped into 32 registers. Thus all the simulations in this chapter assume 20 registers per context for sequential code, and 32 registers for parallel code.

Table 5-1 lists the static and dynamic instruction counts of the different programs.

Benchmark	Type	Source code lines	CSIM instructions	Instructions executed	Avg instr per context switch
GateSim	Sequential	51,032	76,009	487,779,328	39
RTLSim	Sequential	30,748	46,000	54,055,907	63
ZipFile	Sequential	11,148	12,400	1,898,553	53
AS	Parallel	52	1,096	265,158	18,940
DTW	Parallel	104	2,213	2,927,701	421
Gamteb	Parallel	653	10,721	1,386,805	16
Paraffins	Parallel	175	5,016	464,770	76
Quicksort	Parallel	40	1,137	104,284	20
Wavefront	Parallel	109	1,425	2,202,186	8,280

TABLE 5-1. Characteristics of benchmark programs used in this chapter.
Lines of *C* or *Id* source code in each program, CSIM instructions in the translated program, instructions executed by CSIM, and average instructions executed between context switches.

5.2 Named-State Advantages

The Named-State Register file:

- Uses registers more effectively than conventional register files, by only holding live, active data in the register file.
- Supports more concurrent tasks with less register spill and reload traffic than conventional files.
- Uses software and hardware management to run both sequential and parallel code efficiently.

5.3 Performance by Application

The results in this section compare the Named-State Register File to a segmented register file with the same number of registers. The segmented register file consists of 4 frames of 20 registers each for sequential programs, or 32 registers each for parallel simulations. On a register miss, the segmented register file spills and reloads an entire frame of registers, even if they do not all contain live data. The segmented file selects frames to spill using a

Least Recently Used strategy. The NSF is organized as in Chapter 3, a fully associative register file containing 80 or 128 lines of one register each, and a LRU strategy for selecting victim lines.

5.3.1 Concurrent contexts

The average number of contexts resident in a register file indicates the amount of concurrency that an application can maintain in a given register organization. In some cases, the register file limits the number of contexts resident, while in others, the application is unable to generate many concurrent contexts.

Figure 5-1 illustrates the average number of contexts resident in the NSF and in a

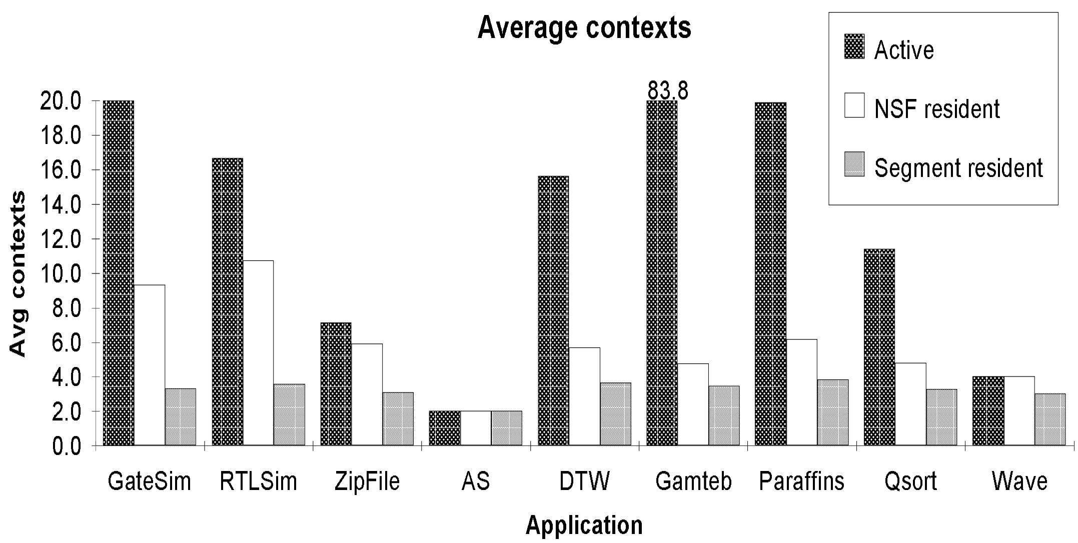


FIGURE 5-1. Average contexts generated per application, and average contexts resident in NSF and segmented register files.
Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

segmented register file. It also shows the average number of concurrent contexts produced by each application.

Among the sequential applications, both GateSim and RTLSim generate many active contexts. These two programs have deep call chains, requiring a context for each procedure invocation. While the segmented register file can hold only four contexts, the NSF holds as many contexts as can share 80 lines. For both of these applications, the NSF holds three times as many contexts as the segmented file.

The ZipFile application, on the other hand, has a relatively shallow call chain, and does not require many concurrent active contexts. Here the NSF register file is able to hold most of the call chain resident at one time. Thus ZipFile will require very little register spilling and reloading on the NSF, while the segmented register file is only able to hold an average of half of the call chain.

Our simulations of parallel programs are hampered by the poor task scheduling of the uniprocessor TAM emulator. (See Section 4.4.4). Since this emulator does not model communication delays or synchronization with other processors, it does not accurately trace the thread execution of fine-grained parallel programs. As a result, most of the Id applications studied here do not exhibit much parallelism. Table 5-1 shows the average number of instructions executed between context switches for each application. But there is a very large variance in each of these numbers. Unlike sequential applications, most of the Id applications intersperse a long series of instructions from one context with a very few instructions from another context.

The parallel programs all have fewer resident contexts than the sequential programs. Some of the programs, such as AS and Wavefront, show remarkably little parallelism. For these two programs, even a four frame segmented register file is able to hold the entire activation tree. DTW and QuickSort spawn somewhat more parallelism, but even these two produce fewer concurrent contexts than a typical sequential application. Only Gamteb and Paraffins generate very deep activation trees.

Even those Id programs which generate ample parallelism do not produce as many resident contexts as a typical sequential application. The reason is that Id programs all touch more registers than the average C program. Figure 5-2 shows the average number of registers accessed per context for each application. The figure shows that while an Id context is resident in a register file, it accesses three times as many registers as a typical C procedure. Since Id contexts consume more space than C contexts, the NSF cannot keep as many resident at one time.

The large number of registers accessed by Id contexts may be a property of Id programs, or it might be solely an artifact of the simulation environment. As noted in Section 4.4, the *TLTRANS* translator folds a very large number of Id local variables into a set of 32 CSIM registers. The translator does not use the number of accesses per variable, or the lifetime of those variables, in mapping them to registers. A true register allocator would do a much better job of mapping variables to registers, and should reduce the number of registers accessed per context. In the absence of a decent compiler, the results shown here only approximate the performance of a real register file.

5.3.2 Register file utilization

One way of measuring the fraction of a register file that is in use is to count registers that contain *live* data. A register is considered live if it has been written by an earlier instruc-

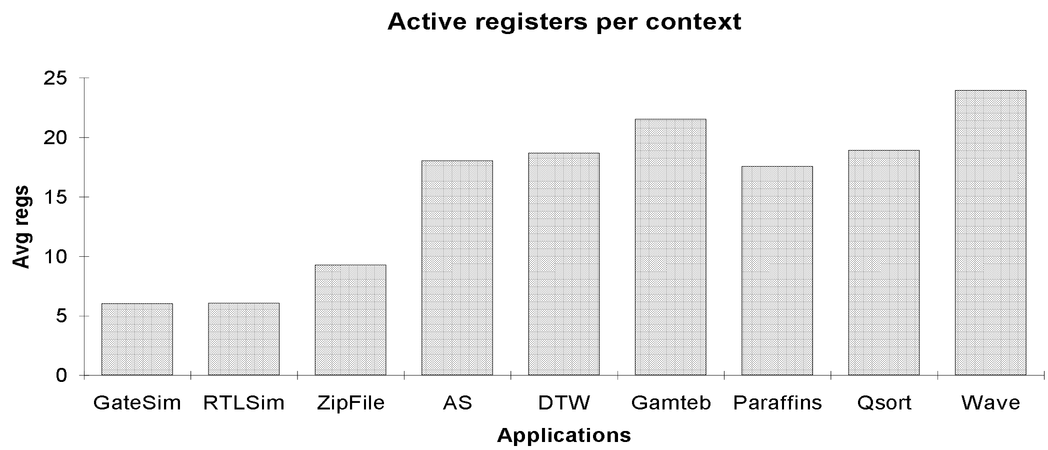


FIGURE 5-2. Average number of registers accessed by a resident context.

tion, and not yet deallocated. But a register that contains live data might be reloaded and spilled from the register file without being accessed by any instructions. A register might contain live data, but there is no reason to load it into the register file unless it will soon be read or written. A better measure of register file usage is the proportion of registers that are *active*, registers that have been read or written since they were loaded into the file.

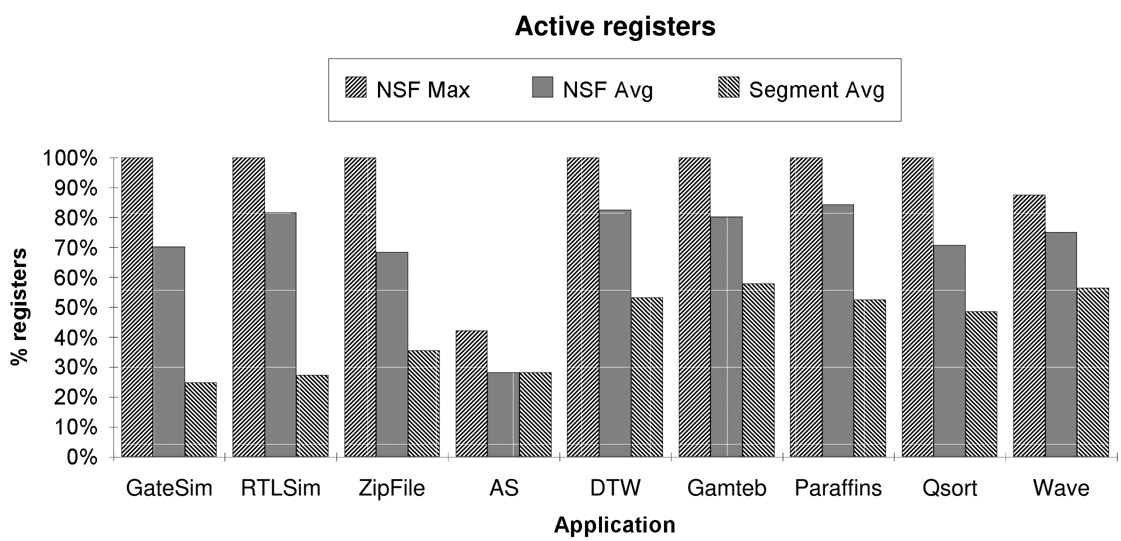


FIGURE 5-3. Percentage of NSF and segmented registers that contain active data. Shown are maximum and average registers accessed in the NSF, and average accessed in a segmented file. Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

Figure 5-3 shows the average fraction of *active* registers in the NSF and segmented register files. It also shows the maximum number of registers that are ever active. Each of the sequential programs generates enough parallelism to fill the NSF register file with valid data. Since sequential programs switch contexts frequently, and touch few registers per context, a segmented register file contains many empty registers. Sequential programs use almost 80% of the NSF registers, more than twice as many active registers as on the segmented register file.

The situation is different for the parallel applications. Each context may touch many different registers, without reusing any of the registers. Some of the registers are initialized when the context is first allocated, and then not touched again until the context is deallocated. The percentage of *live* registers in each context is very high, and the percentage *active* is much less.

While some parallel programs do not generate much parallelism, those that do can fill the NSF with data. These typical parallel programs access approximately 80% of the NSF registers, better than most sequential programs. In contrast, parallel programs are only able to use 50% of the segmented register file.

5.3.3 Register hit rates

Figure 5-4 shows the number of register accesses that missed in the NSF and segmented register files as a percentage of instructions executed. This is greater than the percentage of instructions that were retried, since some instructions miss on several operands.

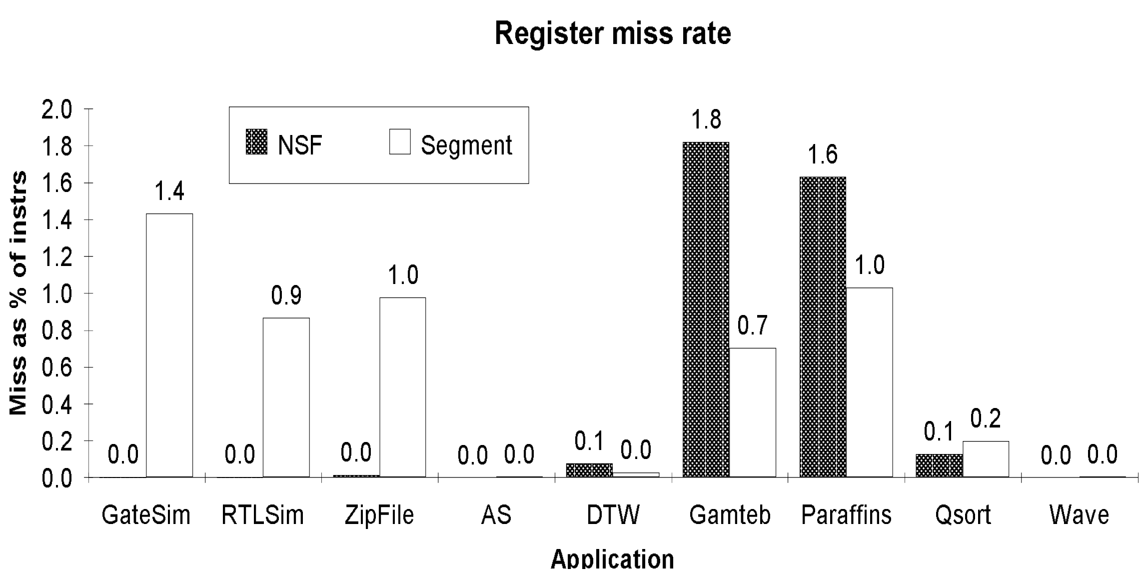


FIGURE 5-4. Register misses divided by total instructions executed.
Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

For all of the sequential applications, the miss rate in the NSF is almost non-existent. While some sequential applications have deep call chains, the LRU replacement strategy and many lines in the NSF allows this file to capture most of the procedure chain. Miss rates in the segmented register file, on the other hand, are two orders of magnitude greater than in the NSF. Building larger segmented register files, which hold more frames, will reduce the miss rate somewhat, but as shown in Section 5.4.1, a segmented register file must be very large to reduce misses to the level of a NSF file.

There is little difference in the miss rates of the NSF and segmented files for AS and Wavefront, the two parallel applications that produce so little parallelism. On the other hand, DTW, Gamteb and Quicksort have twice as many misses on NSF as on a segmented file. This higher miss rate is to be expected, since the segmented file reloads an entire context on a miss, ensuring that successive instructions from this context activation will not miss.

5.3.4 Register reload traffic

The NSF spills and reloads dramatically fewer registers than a segmented register file. Every miss in the NSF reloads a single register, while in the segmented file, it reloads an entire frame. Figure 5-5 shows the number of registers reloaded by NSF and segmented files for each of the benchmarks. This counts reloads of empty lines. Also shown is the number of registers reloaded by the segmented file that actually contained valid data.

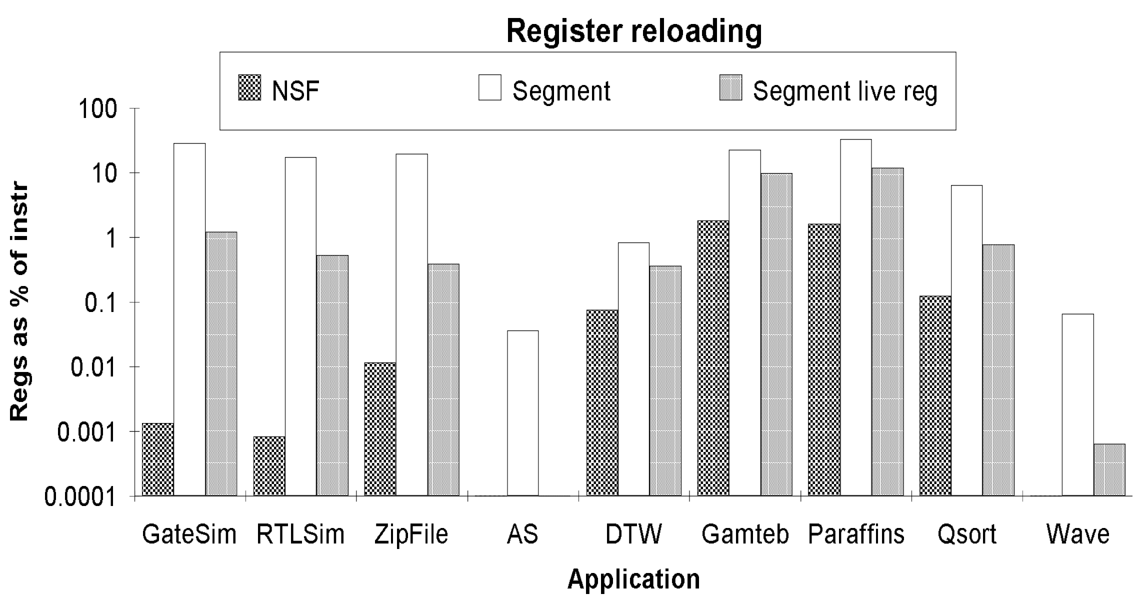


FIGURE 5-5. Registers reloaded as a percentage of instructions executed.
Also registers containing live data that are reloaded by segmented register file. Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

For sequential applications, the segmented register file reloads 1,000 to 10,000 times as many registers as the NSF. As shown in the miss rates of Section 5.3.3, a segmented register file must reload a frame of 20 registers every 100 instructions. Note that if the segmented register file only reloaded registers that contained valid data, it would reduce the number of registers reloaded by a factor of 30. But because sequential programs generate so many procedure activations, and cycle through them so quickly, a segmented register file would still reload 2 to 3 orders of magnitude more registers than the NSF. The NSF does not have as great an advantage over segmented register files in running parallel code. Certainly, the NSF does not reload any registers while running AS or Wavefront applications, which simply do not produce many parallel threads. But for most parallel applications, the NSF reloads 10 to 40 times fewer registers than a segmented file. This is due to the fact that the NSF only reloads registers as needed, while the segmented file reloads an entire frame. But even if the segmented file only reloaded registers that contained valid data, it would still load 6 to 7 times as many registers as the NSF. This is because registers that contain valid data may still not be accessed during the current activation of a context.

This chapter does not show register spill results. But for most applications, registers spilled are the same as those reloaded.

5.4 Experiment Parameters

The results introduced in Section 5.3 compared a four frame segmented register file to a Named State register file of the same size. This section analyzes different design decisions, including the size of the register files, the number of words per line, and spill and reload strategies.

Rather than report separate performance results for each of the benchmarks, this section only shows results for two typical applications. Since the parallel benchmarks behave so differently, no attempt was made to average over the applications. *Gamteb* was selected as a representative parallel benchmark, since it generates enough parallel tasks to fill any sized register file, and the program code is larger than any of the other Id applications. The shorter Id benchmarks are simply toy programs.

For the same reasons, *GateSim*, the largest sequential application, was selected to characterize register file performance on sequential programs. *GateSim* is a large program that runs several hundred thousand instructions.

5.4.1 Register file size

The most basic design decision is the size of the register file, expressed as the number of frames that it can hold. The larger the register file, the more contexts it can hold simultaneously. This translates into lower miss rates and fewer registers spilled and reloaded.

The simulations evaluate segmented files of between 2 and 10 frames long. For parallel experiments, the frames each hold 32 registers, while for the sequential runs, each frame is 20 registers long. For each experiment, the NSF holds the same number of registers as the equivalent segmented file, but organized as single register lines.

Resident Contexts

Figure 5-6 shows the average number of contexts resident in NSF and segmented register files of different sizes. The average contexts resident in a segmented file is linear with the size of the file, but increases slowly. A segmented file big enough for 10 contexts only holds an average of 7 contexts resident. Since the register file only reloads contexts on demand, it fills on deep calls but empties on returns. The larger the segmented register file, the lower its average utilization.

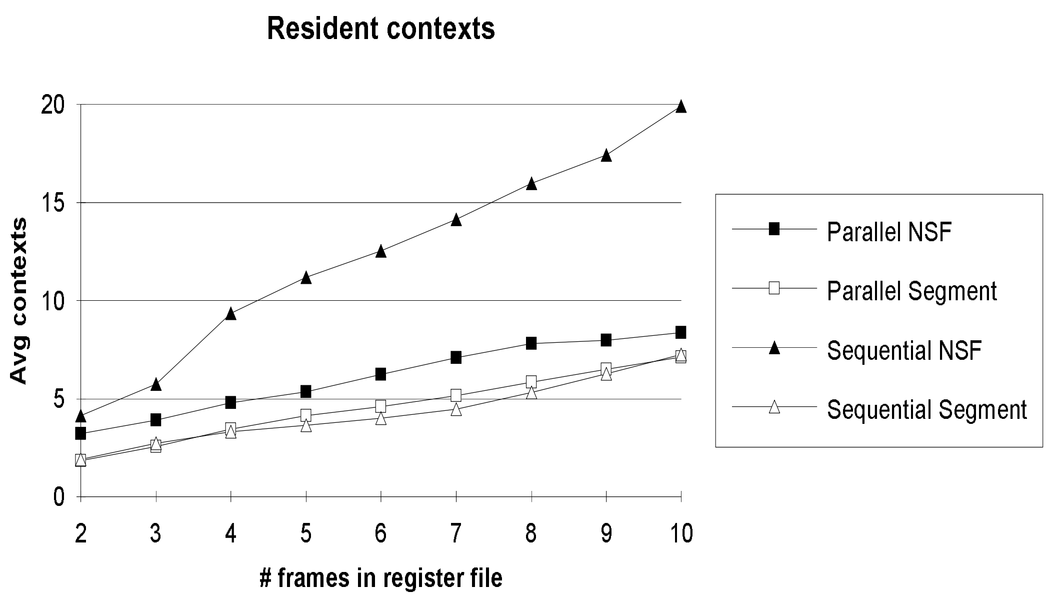


FIGURE 5-6. Average contexts resident in various sizes of segmented and NSF register files. Size is shown in context sized frames of 20 registers for sequential programs, 32 registers for parallel code.

In contrast, the number of active registers per context determines the average contexts resident in an NSF. For sequential programs, which have an average of 6 active registers per context, the NSF holds twice as many contexts resident as there are frames in the register file. Unlike the segmented file, the average utilization of the NSF does not decrease as the register file becomes larger.

However, the NSF only performs slightly better than a segmented file on parallel code, holding one more resident context. A parallel program, which touches 20 registers per context, tends to spill out earlier contexts. As a consequence, the NSF often empties, and register file utilization is lower than with sequential code.

Register file utilization

Figure 5-7 shows the average number of registers that are active in different sizes of segmented and NSF register files. Note that the utilization of segmented files decreases as the size increases. This is because the average number of contexts resident in segmented files is not proportional to the size of the file. Since each context has the same average fraction of active registers, the coarse binding of names to frames reduces the utility of segmented files.

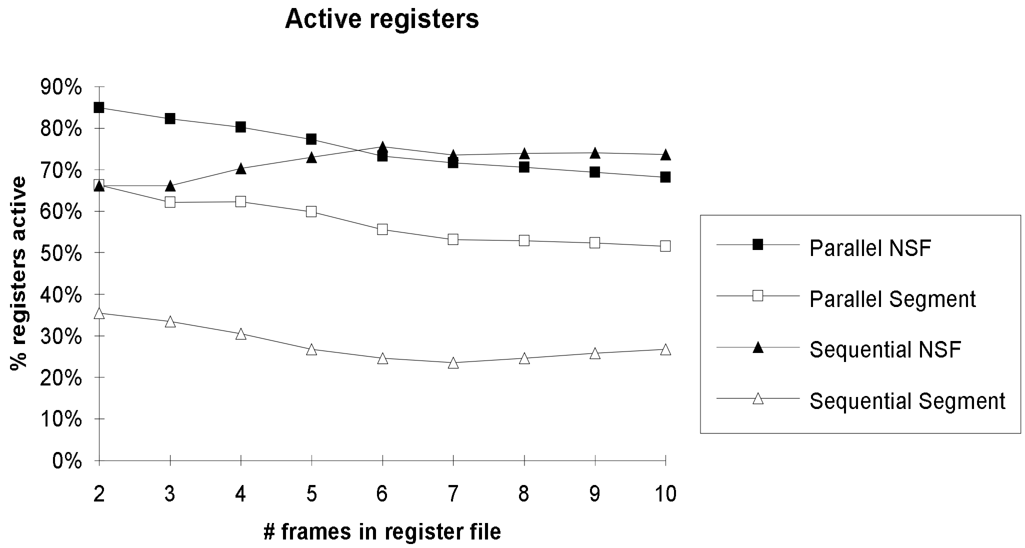


FIGURE 5-7. Average percentage of registers that contain live data in different sizes of NSF and segmented register files.

For sequential applications, which frequently switch contexts and do not touch many registers per context, the NSF touches more than twice as many active registers as a segmented file. And because it can hold a few registers from many different contexts, the NSF has better utilization with larger files.

For parallel applications, the NSF uses 20% more registers than the equivalent segmented file. Since Id applications touch so many registers per context, both register files are less efficient as they become larger.

Register reload traffic

The best argument in favor of Named-State Register Files is the number of register spills and reloads required to run applications. In fact, the smallest NSF requires an order of magnitude fewer register reloads than any practical size of segmented register file.As shown by Figure 5-8, very small segmented files reload a register every 2 instructions for sequential code. That proportion decreases rapidly to a register every 30 instructions for moderate sized segmented files and every 4000 instructions for very large register files.

Parallel programs are more expensive, reloading a register every 8 instructions for average sized files. Even on very large segmented files, parallel programs reload a register every 40 instructions.

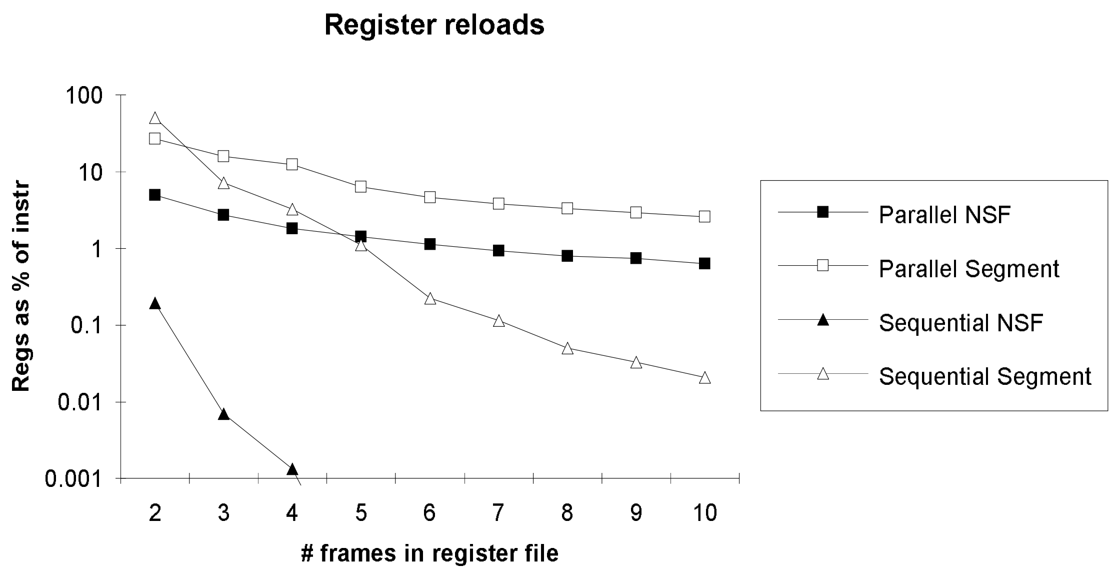


FIGURE 5-8. Registers reloaded as a percentage of instructions executed on different sizes of NSF and segmented register files.

Sequential code running on the smallest NSF, on the other hand, reloads a register only once every 500 instructions. For larger register files, the NSF reloads almost no registers at all. A typical NSF reloads 10^{-4} as many registers as an equivalent sized segmented register file on sequential code.

The NSF does not perform quite as well on parallel code, reloading a register as often as every 20 instructions, and as infrequently as every 150 instructions. This is due to the large number of registers used by each Id context. But the NSF still reloads 5 to 6 times fewer registers than a comparable segmented register file.

5.4.2 Register line width

This section studies the performance of register files with different line sizes, ranging from a NSF with one register per line, to a segmented register file with a line per context. Line size affects register miss and reload rates in several ways. First, larger lines reload more registers on every miss. This can lower the miss rate for larger lines, since subsequent instructions might access registers within the same line. However, it also increases the register reload rate, reloading registers that will not be accessed, or contain no valid data. Finally, as discussed in Section 2.4, in a NSF with single word lines, every write simply allocates a register, without needing to miss or fetch a new line.

Miss ratio vs. line size

Figure 5-9 shows the effect of register line size on read and write miss rates for sequential and parallel applications. Write miss rates behave as expected, showing a gradual decline with increasing line size, as subsequent register writes are more likely to hit in a line that was recently loaded into the file. Of course, the write rate is zero for lines that are a single word wide, since the file need never miss and reload a register that will only be overwritten. Since write misses occur at least 3 times as frequently as read misses, they tend to dominate the register file miss and reload behavior.

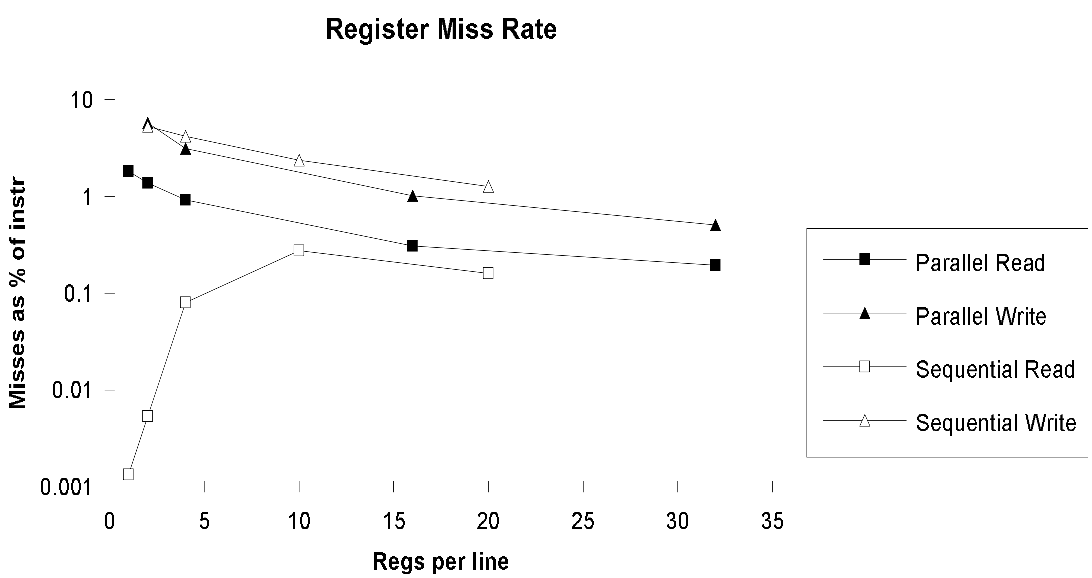


FIGURE 5-9. Register read and write miss rates as a function of line size for NSF register files. Note that register write miss rate is 0 for line size=1. Each file holds 80 registers for sequential simulations, 128 for parallel code.

Note in this analysis, the first write to a line is counted as a write miss, although the line does not yet contain any data. These first writes to a line should just reserve space in the file, perhaps spilling registers, but should not reload any registers. The compiler could ensure this either by tagging allocating writes as described in Section 2.1.3, or by explicitly allocating new lines before touching any registers in that line. While the latter is easy for segmented files with large lines, it is inefficient for small line sizes.

Parallel read misses behave in a similar manner to write misses. For parallel programs, increasing line size reduces the number of register misses, since each context is likely to touch many registers within that line. A line that holds an entire context may reduce the miss rate by a factor of 10 for parallel code.

However, for sequential code, read misses increase dramatically with line size. Segmented register files, in which the line is the size of a context, have 100 times the read miss rate of files with single word lines. For sequential code, in which contexts only touch a few registers between switches, loading many registers on a miss is a liability. Large lines reload registers that are not needed by the current activation, and result in other useful data being spilled out of the file.

Register reloading vs. line size

Figure 5-10 shows the effect of miss rates on register reload traffic. Three different alternatives are shown here. The most expensive scheme does not tag the first write to a new line as discussed above, and blindly loads empty lines. The large number of resulting empty register loads shows that some form of allocating new lines is necessary. A more practical approach tags writes, and does not load any empty lines. Finally, Figure 5-10 also shows the number of registers containing live data that are reloaded by each application. While not all of these live registers might be accessed by the current context, this illustrates how many registers reloaded with each line are empty, and simply waste memory bandwidth.

While most miss rates decrease with increasing line size, each miss is also more expensive. The net effect is that the number of registers reloaded always increases with increasing line size. A sequential program reloads almost no registers for single word lines, and reloads a register for every 30 instructions on segmented register files. Parallel programs require more reloads since each context touches more registers. For parallel applications, small lines reload a register every 20 instructions, while the large lines of a segmented file cause a register to be reloaded every 10 instructions.

This figure also demonstrates the advantages of single word lines in the NSF. On parallel code, an NSF with double word lines reloads 3 times as many registers as an NSF with single word lines. For sequential code, the difference is a factor of 10. This easily justifies the additional cost of single word lines.

Note that a register file with multiple registers per line, which maintained valid bits for each of the registers, could eliminate many write misses and subsequent reloads. The

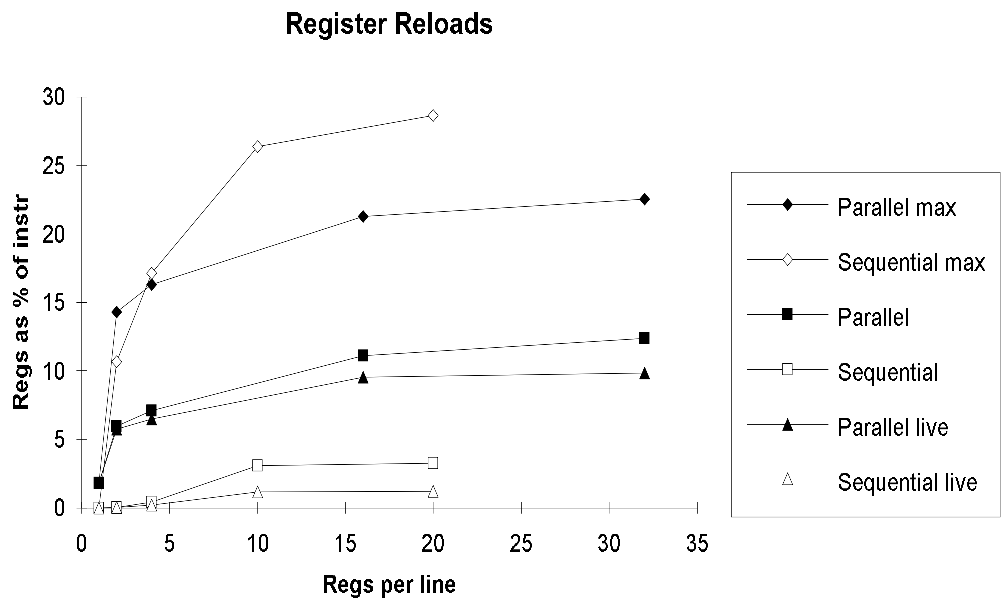


FIGURE 5-10. Registers reloaded on read and write misses as a percentage of instructions. Three sets of experiments are shown here. Maximum values include loads of empty lines on the first write to a new line. Regular experiments do not reload empty lines. Live register reloads only counts those registers that contain valid data. Shown as a function of line size. Each file holds 80 registers for sequential simulations, 128 for parallel code.

reloads due to read misses are a much lower percentage of program run time. The next section discusses this effect in more detail.

5.4.3 Valid bits

The simulations described in this chapter model large register lines by spilling and reloading an entire line at a time. For such an organization, the register file need only tag each line with a valid bit. An alternative organization might group registers into large lines, but tag each register with a valid bit. The address decoders for such a register file organization would be simpler than for a fully-associative NSF with one register per line. However, as shown in Section 3.5, register valid bits and victim selection logic consume almost as much area as a fully-associative decoder.

This section highlights the benefits of tagging each register with a valid bit, for files with large register lines. For such an organization, a register write would never reload a line, but simply allocate the register and set its valid bit. However, a write might spill registers from the file to make room for a new line. Similarly, a register read may cause a register to be reloaded into the file, but would not reload an entire line.

Valid bit hardware is useful in tracking both live and active registers. A register is live if it has been written by a previous instruction, and is active if it will be accessed while the current line is still resident. If the register file only reloads registers as they are needed, it only loads active registers. Otherwise, it could reload an entire line at a time, but only load live registers in the line.

A sophisticated compiler might try to keep track of which registers are valid during each basic block of a program. By loading special trap handlers, or explicitly loading and storing a word of register tags, the runtime software might be able to reduce the number of registers that it spills and reloads. At best, software approaches might be able to spill and reload only live registers. It seems unlikely that a processor could track active registers without a tag on each word in the register file.

Figure 5-11 shows the effect of valid bits on register reloading. The figure compares basic register reloads with live registers reloaded and estimates of active registers reloaded. For parallel code, loading only active registers instead of all registers in a line reduces the reload traffic by as much as one third. The savings is highest for large lines. Loading only live registers accounts for half of that savings. For sequential code, live or active register reloads can be as little as one third of total reloads.

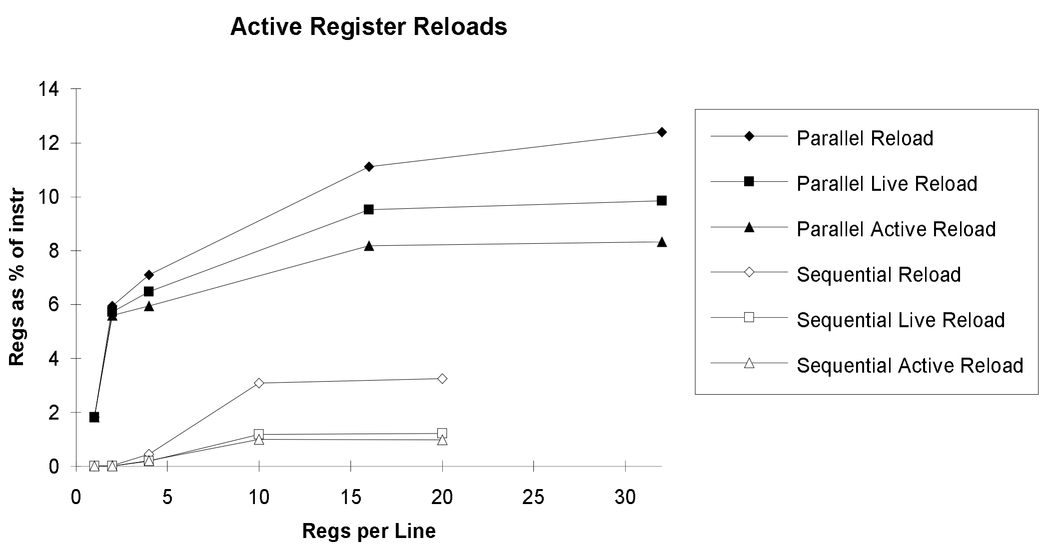


FIGURE 5-11. Registers reloaded as a percentage of instructions.
Basic reloads count registers reloaded on non-empty lines. Live register reloads only counts those registers that contain valid data. Active reloads only counts registers that will be accessed while the line is resident. Shown as a function of line size. Each file holds 80 registers for sequential simulations, 128 for parallel code.

However, an NSF with single word lines is still much more efficient than a segmented register file with valid bits. The NSF requires only 25% of the register to memory traffic required by a tagged, segmented file for parallel code. On sequential code, the NSF reloads 3 orders of magnitude fewer registers than a segmented file with valid bits. By using both valid bits and single word lines, the NSF performs much better than a file using valid bits alone. Since valid bits are a significant fraction of the NSF area, it does not seem worthwhile to build a register file without fully associative decoders.

5.4.4 Explicit allocation & deallocation

As shown in the previous sections, explicit compiler allocation of register lines (allocate-on-write) significantly decreases the number of empty lines reloaded on register writes. Explicit deallocation of individual registers by the compiler should reduce misses and register traffic by freeing registers in the NSF after they are no longer needed.

This study did not perform the compiler modifications required to deallocate registers after their last use. Nor did it include modifications to the register allocator to pack active registers within a single small line.

5.5 Program Performance

Most of the performance results in this chapter express register spill and reload traffic as a fraction of instructions executed by each benchmark. This section attempts to quantify the net effect of different register file organizations on processor performance by counting the cycles executed by each instruction in the program, and estimating the cycles required for each register spill and reload. This analysis is highly dependant on the relative costs of processor instructions and on the design of the memory system. Since these costs are outside the scope of this thesis, the analysis that follows is at best an estimate of true performance.

Table shows the latency in cycles for different instructions and for register file spilling and reloads. Most of these numbers were taken from timing simulations [43] of a Sparc [79] processor, with Sparc2 cache sizes and hit rates. Three different sets of cycle counts are shown: timing for the NSF; for a segmented file with hardware assist for spills and reloads; and for a segmented file that spills and reloads using software trap routines.

Instruction	NSF	Segment	Software
ALU	1	1	1
LOAD	2.1	2.1	2.1
STORE	3	3	3
BRANCH	1.3	1.3	1.3

TABLE 5-2. Estimated cycle counts of instructions and operations.
Compares times for the NSF, a segmented register file with hardware assisted spilling and reloading, and a segmented file with software spill and reload.

Instruction	NSF	Segment	Software
CALL	1	1	1
RETURN	1	1	1
NEWCID	1	1	1
POPCID	1	1	1
Context Switch	2	2	8
Operation	**NSF**	**Segment**	**Software**
Read Miss	0.2	0.2	1
Write Miss	0.2	0.2	1
Register Flush	3	2.3	5.31
Register Reload	2.2	1.8	3.44

TABLE 5-2. Estimated cycle counts of instructions and operations.
Compares times for the NSF, a segmented register file with hardware assisted spilling and reloading, and a segmented file with software spill and reload.

Figure 5-12 shows the resulting proportions of execution time spent spilling and reloading registers for sequential and parallel code. The NSF completely eliminates the overhead on sequential programs, which for a hardware assisted segmented file accounts for 8% of execution time. The difference is almost as dramatic for parallel programs, cutting overhead from 28% for the segmented file to 12% for the NSF.

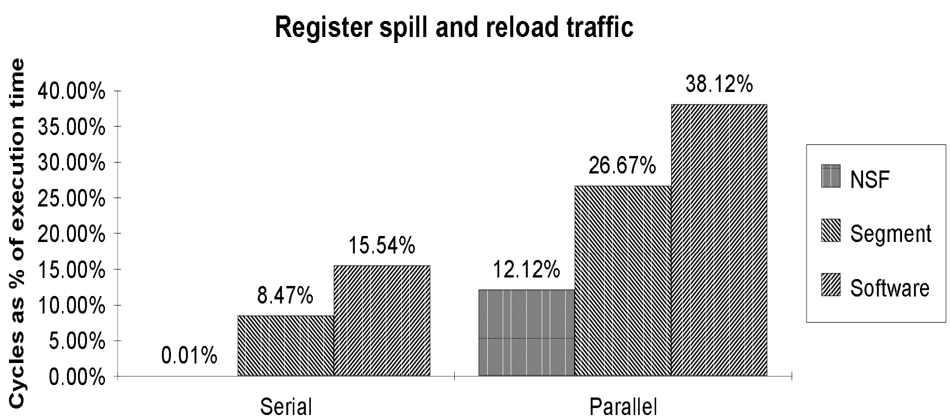


FIGURE 5-12. Register spill and reload overhead as a percentage of program execution time. Overhead shown for NSF, segmented file with hardware assisted spilling and reloads, and segmented file with software traps for spilling and reloads. All files hold 128 registers.

5.6 Results Summary

5.6.1 Goals of the NSF

The simulations in this chapter show that the NSF has met all of its design goals.

Register file utilization

The NSF holds more active data in its register than a conventional file of the same size. The register utilization for both segmented and Named-State Register Files depends strongly on the application. Due to poor task scheduling and register allocation, some parallel applications generate very few contexts. But for large sequential and parallel applications, the NSF fills an average of 80% of the register file with active data. In comparison, the segmented file is only able to keep 25% to 50% of registers busy.

Active Contexts

The NSF holds more concurrent active contexts than conventional files. For sequential code, the NSF holds more than twice as many procedure call frames as a conventional file, without having to spill and reload registers.

Register traffic

The NSF is able to support more resident contexts with less register spill and reload traffic than a segmented register file. Because of the deep call trees and few registers per procedure used by sequential applications, the NSF almost never spills registers. It requires on 0.1% to 1% of the register reload traffic of segmented files. On parallel applications, the NSF reloads an average of 6 to 10 times fewer registers than segmented file.

Fast context switching

Since the NSF has much lower register traffic than a segmented register file and comparable miss rates, it is able to switch contexts more efficiently. For parallel applications, the NSF may miss on twice as many accesses as a segmented file. For sequential programs, the NSF hardly misses on any accesses. The net effect is much lower reload traffic, and the ability to quickly switch between a large number of concurrent contexts.

5.6.2 Alternatives

Large segmented files

One alternative to the high register utilization of the NSF is to build larger segmented register files. But because segmented files are coarsely partitioned, they become less efficient with size. As a result, the NSF holds as many active registers as a segmented register file twice its size. For parallel code, the NSF holds as many active registers as a 20% larger segmented file.

For register reloads, the difference is even more dramatic. A typical NSF file reloads fewer registers than any practical size of segmented file. In fact for sequential code, the NSF reloads 100 times fewer registers than a segmented file twice its size.

Valid bits

Two effects contribute to the performance of the NSF. The first is that when an access misses in the register file, only that register is reloaded, rather than an entire set of registers. This is because each register is tagged with a valid bit. The second is that the NSF is fully-associative with single word lines. Thus the NSF is able to fill the registers with active data, and to hold only most recently used data in the register file.

The experiments in this chapter illustrate that fully-associative, fine-grain addressing of registers is much more important than valid bits on each word. An NSF with single word lines may completely eliminate register spilling and reloading on sequential programs, and cut the reload traffic in half for parallel programs. Single word lines also perform much better than double word lines.

The benefits of valid bits alone are not as significant. A segmented register file with large frames and valid bits for each register may spill and reload 35% to 65% as many registers as a file without valid bits. However, an NSF with single word lines reloads only 25% as many registers as a tagged segmented file on parallel code, and 1000 times less registers on sequential code.

5.6.3 Evaluation

The net effect of the Named-State Register File is significant for both sequential and parallel programs. The NSF eliminates speeds up sequential applications by 8%, in eliminating all register spill overhead. For parallel applications, the NSF cuts the overhead in half, from 26% of execution time with a segmented file, to 12% for the NSF.

Chapter 3 estimated that the NSF may require 30% to 50% more area than a segmented register file. This may amount to 7% of the processor chip area. As transistor budgets increase, and performance becomes more critical, the Named-State Register File is an inexpensive means of improving performance on both sequential and parallel applications.

CHAPTER 6

Conclusion

6.1 Overview

6.1.1 Background

Registers have become a critical resource in modern computer architectures. Virtually all modern processors provide some form of fast, random-access local memory for temporary variables. Alternatives range from top-of-stack buffers [32, 25,11] to vector registers [71] to overlapping register windows [31,67,79]. But by far the dominant form of temporary storage is a small, directly accessed register set [6,82,75].

Several factors are responsible for the popularity of register files:

- High performance:
  Register files are small local memories that are tightly integrated with the processor pipeline. They are typically accessed as simple indexed arrays. This reduces access time and allows them to be easily multiported.
- Short addresses:
  Instructions refer to registers by short offsets. This allows a single instruction to refer to several register operands. The ability to operate on registers efficiently, combined with better compiler technology, has lead to the popularity of modern load-store architectures [34].
- Separate register name space:
  Registers define a region of memory separate from the virtual address space of memory operands[1]. A compiler can manage this name space without regard to access patterns and aliasing among memory allocated data.

However, while these factors encourage the use of registers for temporary variables in high performance processors, they also make it more difficult to manage registers in dynamic and data dependent programs.

Since sequential program performance depends on effective use of the register set, researchers have spent much effort on register allocation strategies [16,17]. Modern compilers are able to allocate registers among basic blocks within a procedure, but it is much more difficult to allocate registers across procedure call boundaries [86,78]. The

1. Unlike stack-based machines [32,11], or architectures with memory-mapped registers [10].

problem is not only to efficiently pack a larger number of variables into a fixed sized register set, but to anticipate the *call graph* of a program at compile or link time. When a number of procedure *activations* compete for a limited register name space, and the order of those activations is not known at compile time, the register space will not be used efficiently. Such a dynamic program will spend a large fraction of its time spilling and reloading registers from memory.

This resource allocation problem is even more severe in multi-threaded environments, when a processor rapidly switches between several concurrent threads [76,47,41]. Parallel programming models use multi-threading to cope with long communication latencies and frequent, unbounded synchronization delays. When programs can dynamically spawn parallel procedure activations across multiple processors, it is impossible to statically predict a thread schedule.

Existing parallel processors deal with dynamic scheduling either by not using registers at all [65,21], by not preserving registers across synchronization points [42], or by segmenting the register space among a few resident threads [76,47,4]. Since effective use of registers can dramatically improve program performance, processors need a form of register file that speeds both statically scheduled code and dynamic context switching.

6.1.2 Contribution of this work

This thesis introduces the Named-State Register File, an associative, fine-grain register structure. The NSF uses registers more effectively than conventional register files, and supports a large, dynamic set of local variables with less traffic to memory.

The NSF uses a combination of compile-time and hardware register management to map variables into a limited physical register set. The NSF is a fully-associative structure with small register lines, to bind variables to register locations at much finer granularity than conventional register files. The NSF provides a large register name space by appending a *Context ID* to each register reference.

This thesis shows an implementation of the NSF, and analyses its access time and chip area. The thesis also simulates a number of register file organizations running both sequential and fine-grained parallel code, to evaluate the performance of the NSF relative to conventional register files. This research has shown that the NSF can significantly improve program performance for a variety of programming models.

6.1.3 Scope of this work

The Named-State Register File provides a mechanism for a large number of concurrent *activations* or *contexts* to share a register name space. The NSF does not impose any policy on how those contexts share the registers. The activations may be executed in any order. The NSF spills and reloads registers from backing store as needed.

Depending on how a programming model defines a *context*, context switching may take several forms. Some common examples of context switching are:

- A process switch in a multiprogrammed operating system.
- Light-weight task switching in user applications [3].
- A procedure call in a sequential language.
- Processor multithreading to handle I/O [80].
- Block multithreading among several concurrent threads [29,2].
- Cycle-by-cycle multithreading among a few threads resident on a processor [76,47].

While the NSF can efficiently support most of these forms of context switching, it performs best when switches are unpredictable, and the time between switches is comparable to the cost of loading and unloading a context. It would be inefficient to allocate a separate Context ID for each user process in an operating system that switched processes every few milli-seconds. This would consume CIDs unnecessarily, and any benefit would be hidden in the time required to switch between such heavy-weight processes.

On the other hand, as shown by the simulations in this thesis, it is easy to compile code which allocates a new CID for each procedure activation. The NSF can hold a significant fraction of a sequential call chain without spilling and reloading registers. An application might combine context switching on procedure calls and among concurrent tasks. By assigning CIDs among the tasks, the processor could run each task with efficient procedure calls, and also rapidly switch between tasks.

Finally, an application might use several contexts within a single procedure. Since allocating and deallocating contexts is inexpensive in the NSF, this provides a mechanism for expanding the range of registers accessible to a single procedure. It also simplifies the task of register allocation across basic blocks. The penalty for allocating additional contexts is low, since only active registers will be assigned space in the NSF.

6.2 Summary of Results

6.2.1 Implementation

The area of an NSF relative to conventional register files depends on its organization. A fully-associative Named-State Register File with single word lines is 50% larger than a conventional register file with the same number of registers. This increases a typical processor's chip area by less than 5%. Much of the additional area for the NSF is devoted to valid bits and logic for handling register misses, spills and reloads.

Many modern processors architectures have introduced wide data words [57], and additional ports into the register file for multiple functional units [24]. Both organizations should reduce the relative area devoted to associative decoders and spill logic in the NSF. The NSF's fully-associative address decoders consume only 15% of the register file area, and increase its access time by only 6%. A Named-State File should only be marginally larger than conventional register files for these advanced organizations.

6.2.2 Performance

The small register lines of the Named-State Register File lead to much better utilization of the register set for typical applications. Small lines also significantly reduce the register traffic required to support these applications. For large sequential applications, a moderate sized NSF is able to capture most of the procedure call chain. As a result, the NSF spills and reloads only 1/4000 as many registers as a conventional register file of the same size. On sequential code, the NSF has less register traffic than a conventional file with 4 times as many registers.

The NSF's behavior on parallel applications is highly dependant on efficient register allocation. For the applications studied here, which did not allocate registers effectively, the NSF reloaded 1/6 as many registers as a conventional register file. For these parallel programs, the NSF has less register traffic than a conventional file with more than twice as many registers.

Most of the performance benefit of the NSF is because it is fully-associative at the granularity of a single register. Conventional register files may use explicit register allocation or valid bits on individual registers to allow sub-block reloading and reduce traffic to memory. But an NSF with valid bits and single word lines outperforms register files with valid bits and large lines. The NSF requires only 1/4 the register traffic of the best segmented register file on parallel code, and only 1/1000 the register traffic for sequential programs.

Finally, while the effect of register reloading on program performance depends on the processor's instruction set and memory system, this study estimates that the NSF speeds execution of parallel applications by 17% to 35%, and of sequential applications by 9% to 18%.

6.3 Future Work

Initial studies and simulations indicate that the NSF has potential to significantly improve the performance of dynamic sequential and parallel programs. However, a number of issues in software to exploit register file organizations were not addressed by this thesis.

Register allocation

The programs simulated in this thesis were compiled with very simple register allocation policies. In fact, as discussed in Section 4.4.4, the Berkeley TAM compiler used to compile parallel Dataflow applications did no register allocation at all [72]. This severely compromised the performance of those applications. It is hoped that a more realistic register allocator would use fewer active registers per context. For such code, the Named-State Register File would use registers much more efficiently, and would perform much better than conventional register files, as indicated by its behavior on sequential applications.

The NSF has the ability to explicitly allocate and deallocate individual registers. None of the simulations in this study explicitly deallocated registers after their last use. While freeing up register space in the NSF could improve register utilization and reduce spill traffic, it is unclear what percentage of registers could be explicitly deallocated in this manner. Conventional register usage analysis must often apply conservative estimates of register lifetimes. Further study is needed to investigate the efficacy of register deallocation algorithms and their effect on NSF performance.

Another register allocation issue is how to group active registers within fixed-sized lines. The simulations in this thesis compared register files with different line sizes and measured the register spill traffic they required. An NSF with single register lines was found to perform much better than files with larger line sizes. But these simulations all used the same register allocation strategy. If a compiler could target its register usage strategy to a file with a particular line size, it might be able to group active registers together in a single line, so several active registers are reloaded at a time [85]. Since memory design often supports pipelining and block reloads, this might make Named-State Files with line sizes of two or four registers more efficient than those with single register lines. However, it is not clear how well a compiler could operate with these additional constraints.

Thread scheduling

As discussed in Section 4.4.3, Dataflow applications simulated in this study were traced by a simple, uniprocessor emulator. The emulator did not realistically portray how Id programs might run on a large-scale parallel processor. The resulting thread trace does not accurately model the effect of communication latency, synchronization between processors, and data distribution across a multicomputer. To evaluate the effect of an NSF on parallel processor performance, the simulations in this thesis should be re-run with actual thread traces from those parallel machines. Preliminary results indicate that Id programs switch contexts much more frequently on multiprocessors than on a single processor [21]. This suggests that the NSF can significantly improve the performance of large-scale parallel processors.

Context management

The NSF uses Context IDs to identify concurrent procedure or thread activations. As discussed in Section 6.1.3, there are many ways of allocating CIDs among a set of activations. One alternative that was not investigated in this thesis was to allow a single procedure or thread activation to use several distinct CIDs. This provides a flexible method of assigning large numbers of registers to a particular context, without burdening all contexts with large register sets. It also may simplify register allocation among basic blocks in a procedure, especially for parallel code in which those basic blocks may run in any order [72].

In order to take advantage of the larger register space defined by Context IDs, an application must manage those CIDs among the active contexts. There are a limited number of CIDs, and either procedure call code or thread schedulers must assign them to dynamically created contexts. While the simulations in this thesis indicate that a small set of CIDs is large enough to support many contexts without conflict, this deserves further study. In particular, simulating specific scheduling algorithms would ensure that a small set of CIDs does not interfere with efficient thread scheduling.

The preceding sections illustrate the importance of the interaction between hardware and software in computer architecture. This thesis has proposed mechanisms that use a combination of hardware and software to efficiently manage processor resources. This combination exploits static knowledge of program structure and adapts well to dynamic runtime behavior.

APPENDIX A
A Prototype Named-State Register File[1]

David Harris
September 24, 1992

A.1 Abstract

This paper summarizes the work that I have done on the Named-State Register File project as a UROP over the summer of 1992. It describes the functionality of the Named-State Register File and the objectives of this project. Then it discusses implementation details at both schematic and layout levels, including a pinout diagram of the chip that I have sent for fabrication. It details the functionality of the chip that has been sent for fabrication, the timing, testing, and pinouts, and concludes with a set of preliminary results.

A.2 Operation

Register 0 of every context ID is always present in the register file and is hardwired to 0, as is the convention in many processors. Write operations to this register are ignored, making it a suitable write address for operations such as NOP.

When a write operation occurs, the register is allocated if it does not already exist or is not currently in the physical register file. In order to guarantee that writes will succeed, the Named-State Register File always maintains one empty physical register. If necessary, another register may be flushed from the cache to make room. This is done by halting the processor pipeline and performing a special read cycle to read the victim out of the register file and directing the CPU to write it to main memory for storage, at the same time invalidating the register in the cache.

When a read operation occurs, read request lines may flag that either of the two read operations failed because the desired register is not currently in the cache. If this happens, the processor must fetch the registers from storage in main memory.

Three additional control lines allow tasks to deallocate registers when they are no longer needed. Either of the two registers read may be freed at the end of the read. In addition, the

1. This appendix was extracted from Internal Memo 46 of the MIT Concurrent VLSI Architecture Group and slightly edited for inclusion in the thesis.

entire set of registers used by a given Context ID may be released at once when the process using that ID terminates.

A.3 Objectives

The objective of this project was to design and build a Named-State Register File to verify that the idea is efficiently realizable and to obtain estimates about timing and chip area required.

I designed a Named-State Register File with the following specifications: The register file contains 32 physical 32 bit registers. The file supports two reads and a write every cycle; each register address is a 10 bit quantity consisting of a 5 bit offset and a 5 bit context ID (CID). The entire circuit should run as fast as possible and consume as little area as possible. In addition, the circuit should scale to 128 physical registers without requiring complete redesign.

Peter Nuth and Bill Dally provided advice on a variety of implementation problems and Whay Lee and Noble Larson gave me great help with the fabrication process.

A.4 Circuit Description

The circuitry in the Named-State Register File is moderately complex. A series of figures at the end of this document show the complete details of the design. This section provides an overview of the entire chip's logic, then delves into timing, naming conventions, a precise description of the Named-State Register File functionality, and schematic-level description of each cell.

A.4.1 Overview

Figure 1 shows a block diagram of the entire Named-State Register File. At this level of detail, it is very similar to a conventional register file: the only difference is that the simple address decoder of a conventional file is replaced by more sophisticated logic to maintain the cache information. As always, there are three data busses: a 32 bit write bus and two 32 bit read busses. Also there are three 10 bit address busses specifying the offsets and Context IDs for read and write operations. The remaining control lines are specific to the Named-State Register File operation. A Reset line clears all of the valid bits in the cache, initializing the file to an empty state and marking register 1 as the first victim to flush. Three free lines control deallocating registers when they are no longer needed. The Flush-Request line indicates that the cache is full and directs the processor to stall the pipeline while the current victim is flushed from the file. The two ReadRequest lines indicate that a read operation missed and that the processor must fetch the desired operand from main memory before continuing execution of the given process.

A.4.2 Clocking

The Named-State Register File is driven by a two phase non-overlapping clock. During Phi1, the register file lines precharge while the Decode/Match logic performs its dirty work. During Phi2, the register file performs the appropriate read and write operations while the cache logic precharges in anticipation of the next phase. Also during Phi2, the allocate logic chooses the next empty line to write.

A.4.3 Naming Conventions

Some naming conventions reduce confusion given the number of signals and cells in the Named-State Register File. All names begin with an uppercase letter and each succeeding word in a name also begins with an uppercase character. The cache logic is divided into seven main functional units. The name of each unit ends with the word Block. Each block consists of many rows of cells containing logic, one for each register in the file. The name of each logic cell ends with the word Cell. There are a number of large amplifiers for driving signals down long lines. The name of each of these cells ends with the word Driver.

The three ten-bit address busses are named R1[i], R2[i], and W1[i] ($0 \le i < 10$). The three 32-bit data busses are named Read1[i], Read2[i], and Write[i] ($0 \le i < 32$). There are four input lines, Reset, FreeR1, FreeR2, and FreeCID. There are three outputs, FlushRequest, ReadRequest1, and ReadRequest2. The non-overlapping clock signals are named Phi1, Phi2, Phi1Bar, and Phi2Bar.

In addition, there are a number of internal signals, shown in Figure 2 on the overview of the Decode/Match circuitry. Each of these signals is used on every row of the logic; in a 32 register Named-State Register File, a signal named MeltDown will have instances MeltDown[i] ($0 \le i < 32$). MeltDown[0] corresponds to the bottom row, while MeltDown[31] corresponds to the topmost row. There are six match lines used by the programmable decoders: R1OffMatchBar, R1CIDMatchBar, R2OffMatchBar, R2CIDMatchBar, WOffMatchBar, and WCIDMatchBar. These match lines are used to compute three word lines: R1Word, R2Word, and WWord. In turn, these word lines control three lines which drive the register file: R1Out, R2Out, and WOut. The R2CIDMatch line is also derived from the R2CIDMatchBar signal. There are two valid lines, ValidBar and ValidA. The FlushMe line is used on flush cycles and the Allocate line chooses the next row in which to perform a write operation.

A.4.4 Functionality

Now we can precisely describe the functionality of the Named-State Register File and the responsibilities imposed upon a processor which uses this Named-State Register File.

The Named-State Register File contains N (in this case N=32) rows of Decode/Match logic and registers. Row 0 is hardwired to always be present and match on Offset 0 of any CID. Each row contains a valid bit which determines if the register on that row contains valid data and a programmable decoder which matches the virtual address of the register stored on that row. A shift register running the height of the cache (except row 0) chooses the next victim for flush operations. A tree structure can identify in logarithmic time the first free line to allocate for write operations.

To initialize the cache, the Reset line must be raised for two consecutive cycles. This clears all of the valid bits and sets the victim select shift register to point at row 1.

Let us begin by examining the two phases of an ideal cycle where both read operations succeed and the cache is not close to full. During Phi1, the bitlines of the register file portion are precharged for read operations and properly set up for write operations. The Decode/Match logic matches both read addresses to raise the appropriate R1Word and R2Word lines. The write function is slightly more complex. Because the Named-State Register File does not know in advance if the write address is already in the cache, it must perform a write on both the actual register (if the register is present) and on another empty line which should be allocated in the event that the desired register was not already in the cache. The programmable decoder on the allocated line latches the write address and the valid bit of that line is initially set on the assumption that the register was not already in the cache. During Phi2, the valid bit of the newly allocated line is cleared if it is found that the write address already was in the cache. Also, the topmost empty row is chosen for the next allocate operation and the cache logic is precharged again. Meanwhile, the register file performs the desired read and write operations.

There are a few less-ideal situations where extra processing must occur. When the last empty row is allocated (even if it is not actually needed), the Named-State Register File must flush a valid row (called the victim) from the cache to guarantee an empty row for the next write operation. This is done by raising the FlushRequest signal near the start of Phi2. During the next cycle, the processor should stall the pipeline. The cache reads the victim onto the Read1 bus and clears the valid bit in the row that contained the victim. The processor must write this value to an appropriate location in main memory to store it until needed in the cache again. This also bumps the victim select shift register up by one row until it reaches the top of the column at which point it wraps around to row 1 again. In order to perform a flush operation, the address of the flushed line must be available, something not done in this design due to space limitations. This can be done either by running a tap out of each programmable decoder or by adding an additional ten bits to each register file to contain the address of the register as well as the data contained.

If a read operation misses, a ReadRequest line is raised by the end of Phi1. The processor must restore the missed register from main memory before continuing execution of this process. Typically, this may be done by sending a memory request off to a memory unit, then performing a context switch and continuing execution of another process for a while.

Care must be taken that this other process is not also waiting for a failed read. When the register's contents are fetched from main memory, the processor should stall the pipeline and insert a special cycle to write the fetched value into the register file. Then the instruction that caused the miss should be re-executed.

When a process is done with a register, it should raise the appropriate free line on the last read of that register. This clears the valid bit of that register. When the process terminates, it should raise the FreeCID line. This releases all of the registers with the same context ID as that used in the R2 address.

A.4.5 Cell Descriptions

Figure 2 illustrates how the various blocks in the Decode/Match logic communicate. This section describes each of those blocks in more detail.

The DecodeBlock (Figure 3) matches the virtual address of various registers using a programmable decoder. Five columns match the offset; another five match the CID. When a register is initially allocated, the decoder latches for that line are loaded with the write address. On future reads and writes, all of the decoder latches must match the address in order to pull down the match lines that indicate that the register may be in the cache (the valid bit must also be set to indicate that the register has not been flushed but not reused yet). The match lines are in effect a pair of six transistor precharged NAND pulldown chains; offset and CID are matched separately and combined in the GateBlock to prevent the excessive delays from a 11 transistor pulldown. The offset and CID match cells must be interleaved; otherwise, one block will perform faster and the other slower due to uneven capacitance distribution. Note the danger of charge sharing; a hefty capacitive load must be present at the right end of the array to prevent the output from accidentally being brought low via charge sharing. Note also that row 0 of this logic should be hardwired to match offset 0 of any CID.

The GateBlock (Figure 4) consists primarily of NOR gates to drive a word line if the offset and context ID of the address match and the valid bit is set on that given word line. An extra gate is required for handling flushes to drive the R1Word line corresponding to the line to flush.

The MissBlock (Figure 5) has three lines to detect misses on the read or write addresses. If a read fails, the appropriate ReadRequest line is raised to ask the processor to fetch the missing register; if a write address is missed, a new line is allocated during the same phase for the new register. Note that the block is structured as a N-bit precharged NOR gate with latches at the bottom to record the result when it is valid by the end of Phi1.

The ValidBlock (Figure 6) contains some messy ad-hoc logic to set and clear a valid bit latch. On cycles where a register should be flushed or the cache is reset, the valid bit must be cleared. During Phi1 of every cycle, the topmost empty line is allocated; if the write succeeded that cycle, the line is then deallocated during Phi2. Two more latches sample the valid bit at appropriate times to insure valid values when they are needed elsewhere in the circuit. Note that row 0 should be hardwired to always be valid. Some equations might clarify this:

ValidLatch set on:		
	Allocate • Phi1	
ValidLatch cleared on:		
	FreeR1 • R1Word • Phi1	or
	FreeR2 • R2Word • Phi1	or
	FreeCID • R2CIDMatch • Phi1	or
	FlushMe • Phi1	or
	WHit • Phi2 • Allocate	
ValidA = ValidLatch sampled during Phi1		
ValidBar = ValidLatchBar sampled during Phi2		

TABLE A-1. Valid bit logic.

The VictimBlock (Figure 7) is a shift register that chooses the next victim to be flushed when the Named-State Register File fills. At reset, a single 1 is loaded into the shift register in row 1. During flush cycles, the register cell containing the 1 raises the FlushMe line for the corresponding row to flush that register and the shift register advances by one upward, looping back when the top is reached. Row 0 is never a candidate to flush. True LRU (Least Recently Used) replacement policy of course would be somewhat more efficient, but this approach uses a minimum of chip area.

The FullBlock (Figure 8) determines when the cache is full. It is built from a N-input precharged NOR gate with a latch at the bottom to grab the FlushRequest signal at the end of Phi1. It is possible to request a flush even when one line is still empty if that line is allocated on the given cycle but not actually needed.

Finally, the AllocateBlock (Figure 9) is a binary tree structure used to choose the topmost empty line to allocate. It is built from a chain of alternating positive and negative logic. I have considered a number of other designs including 4:1 cells and variants on the Manchester carry chain, but simulations show that this circuit performs better than the alternatives that I have explored.

A.4.6 Critical Paths

Addresses are presumed to be stable at the DecodeBlock throughout Phi1. During Phi1, the decoders must pull match lines low. These match lines must be combined with the valid bits in GateBlock to control the word lines. The word lines must be latched by the end of Phi1. Also, the miss lines and valid bits may be cleared by word lines and must be ready to be latched at the end of Phi1. This sets a lower bound on the length of the Phi1 cycle.

$$\text{Phi1} \rightarrow \text{Match} \rightarrow \text{Word} \rightarrow \begin{cases} \text{Latch} \\ \text{Valid} \rightarrow \text{Latch} \\ \text{Miss} \rightarrow \text{Latch} \end{cases}$$

Register access dominates the Phi2 cycle. The word lines must be driven across the register file. The proper registers must respond, either being written or being read onto the bitlines. The two read bitlines must be latched by the end of Phi2 so that they may be precharged again during Phi1.

$$\text{Phi2} \rightarrow \text{WordDrivers} \rightarrow \begin{cases} \text{ReadRegister} \rightarrow \text{Latch} \\ \text{WriteRegister} \end{cases}$$

Of course, precharge of the large NOR structures in the decoder must also occur during Phi2, but this is unlikely to be a bottleneck until N becomes very large.

The other critical path is the selection of the next empty line to allocate. At some point during Phi1, the valid bits (presuming need for of the last allocate line) become valid. The ValidA signal must propagate up and down a tree in the AllocateBlock. This may be a slow step because the tree has a depth proportional to the logarithm of the number of registers (i.e., 5 for 32 registers, 7 for 128 registers). Finally, it must be latched by the end of Phi2 for use in the next cycle.

$$\text{SetValid} \rightarrow \text{AllocateTree} \rightarrow \text{Latch}$$

A.5 Layout Considerations

The Named-State Register File was implemented in a standard 2 micron N-well CMOS technology. I used the GDT tools to lay out and simulate the cache. While I can make no claims that the layout is the best possible realization of the Named-State Register File's functionality, I did make my best effort given the time constraint of one summer to minimize chip area and optimize for speed.

The Named-State Register File fits sideways in a 4600x6800 micron frame. Metal level 1 lines carry signals along the rows; metal level 2 lines bring VDD and GND, clocks, address lines, and control lines vertically down blocks. Typically, the bottom half of each cell contains the metal 1 signal busses and the top half contains logic. Each row is 83 microns tall. The largest arrays of logic are the registers themselves (32 columns, each 70 x 83 microns) and the decoder cells (10 columns, each 171 x 83 microns).

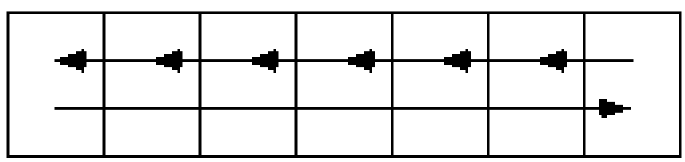
After completing a significant amount of layout, I revised the design to eliminate an extra line, formerly called ValidF. The ValidF logic is still in the Valid cell and the unused line still runs through the Victim and Full cells; in a redesign, some space may be saved by removing the unneeded line and utilizing the freed space.

A.6 Scaling Considerations

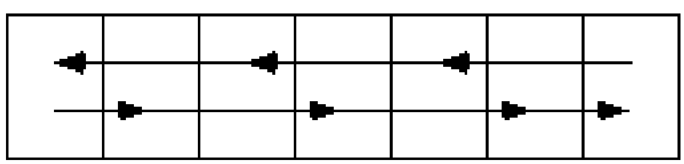
Some effort was made in designing the Named-State Register File to scale well to larger sizes of register files (such as 128 registers). However, there are a number of changes that must be made to maintain performance after scaling.

On account of the capacitance of large bit lines, the register file itself degrades in performance as the number of registers goes up. To compensate, it is standard practice to place sense amplifiers on the bit lines to measure small changes in voltage on reads and more rapidly determine the output of the file. The register file is designed with complementary read lines already to facilitate adding sense amps.

The victim select shift register has a lengthy wire from the topmost row wrapping around to row 1. It is important to simulate this register carefully and determine that the signal can propagate all the way from row N-1 down to row 1 during Phi2. If this becomes a significant problem, one can rearrange the shift register by interspersing up and down shift registers so that every shift operation only has to propagate by the height of 2 rows.



Standard Shift Register



Interleaved Shift Register

The FlushRequest and ReadRequest pulldown chains are likely to be a limiting factor in the timing of very large register files. For instance, for N=128, the chain is effectively a 128 input NOR gate. It may be necessary to rearrange this as a logarithmic tree of NOR gates to overcome the tremendous capacitance problems and large amount of power dissipated by the precharged logic.

A number of large buffers are used to drive signals down the columns. These buffers will have to be scaled to efficiently drive the larger capacitances of longer lines.

Finally, electromigration may become a serious problem for larger Named-State Register Files. The current design just uses very rough estimates of currents and duty cycles to size the clock and power supply lines. For a practical system, these numbers should be given more consideration lest the circuit tear itself apart during normal usage.

A.7 Timing Simulations

To get some information on cycle times, I ran a large simulation of the entire chip using Lsim. Unfortunately, Lsim is notoriously bad for obtaining accurate results; I have typically observed results 50% to 150% slower than Spice produces for the equivalent circuit.

I measured the following numbers from Lsim. Each includes the delay through a three-inverter buffer to drive the potentially long line from the cache to the output pads.

From Phi1 to:		
	ValidA rises on allocated line:	9.1 ns
	R2CIDMatch:	10.5 ns
	R2OffMatchBar:	11.3 ns
	R2Word:	16.3 ns
	ReadRequest2Bar falls:	22.8 ns
From Phi2 to:		
	R2Out:	4.0 ns
	Read bitlines pulled low:	5.4 ns

TABLE A-2. Simulated signal delays from each clock phase.

The timing on the read bitlines appears suspiciously fast to me; possibly the measurement was bad. Clearly, the time for the register array during Phi2 is much faster than the time for the Named-State Register File logic. I was unable to measure the timing to compute FlushRequest with the outputs available; however, it is a function of ValidA, so has at least 13 ns to be computed. I also was unable to measure the time to compute the next Allocated line, but that can occupy all of Phi2, so it should have plenty of time. Therefore, computing ReadRequest is probably the critical path. Assuming 24 ns phases and 1 ns of non-overlap, the Named-State Register File can run at 20 MHz in 2 micron technology.-Given the inaccuracies of Lsim, I would expect that the actual chip would run at between 20 and 40 MHz.

A.8 Pinouts

I designed and carefully simulated a 4 row Named-State Register File. For the final chip, I generated a 32 row cache with 32 bit registers. In addition, this version contains the various drivers for the columns and address lines. This logic has not received the same amount of simulation due to time constraints and is a potential source of critical failures.

The completed logic was placed in a 64 pin pad frame and a number of inputs and outputs were connected in order to perform some tests and timing of the actual chip. Inputs are connected directly to the column drivers. Outputs use a 3 stage buffer to drive the capacitance of the lines going to the pins; therefore, all outputs are actually the complements of the signal. Finally, there are three VDD pins, three GND pins, four clock pins, and three pins for testing very simple circuitry.

Only a few of the 30 read and write address pins were connected; all the rest are tied to 0. Likewise, only a few of the 96 data lines are connected. All of the major input and output control lines are available; in addition, a few of the match, valid, and flush lines are tapped for timing information. Figure A-1 below shows the pinouts of the chip:

A.9 Epilogue

The prototype chip was fabricated by MOSIS, and we received first silicon in October ‘92. In spite of our best efforts, the chips did not function. The pad frame used for this prototype contained numerous errors, including hard power to ground shorts. This pad frame was donated by another group and imported into our CAD tools for this project. The GDT tool’s design rule checker was unable to verify the pad frame prior to fabrication. As a result, we have been unable to test any logic on the prototype chip.

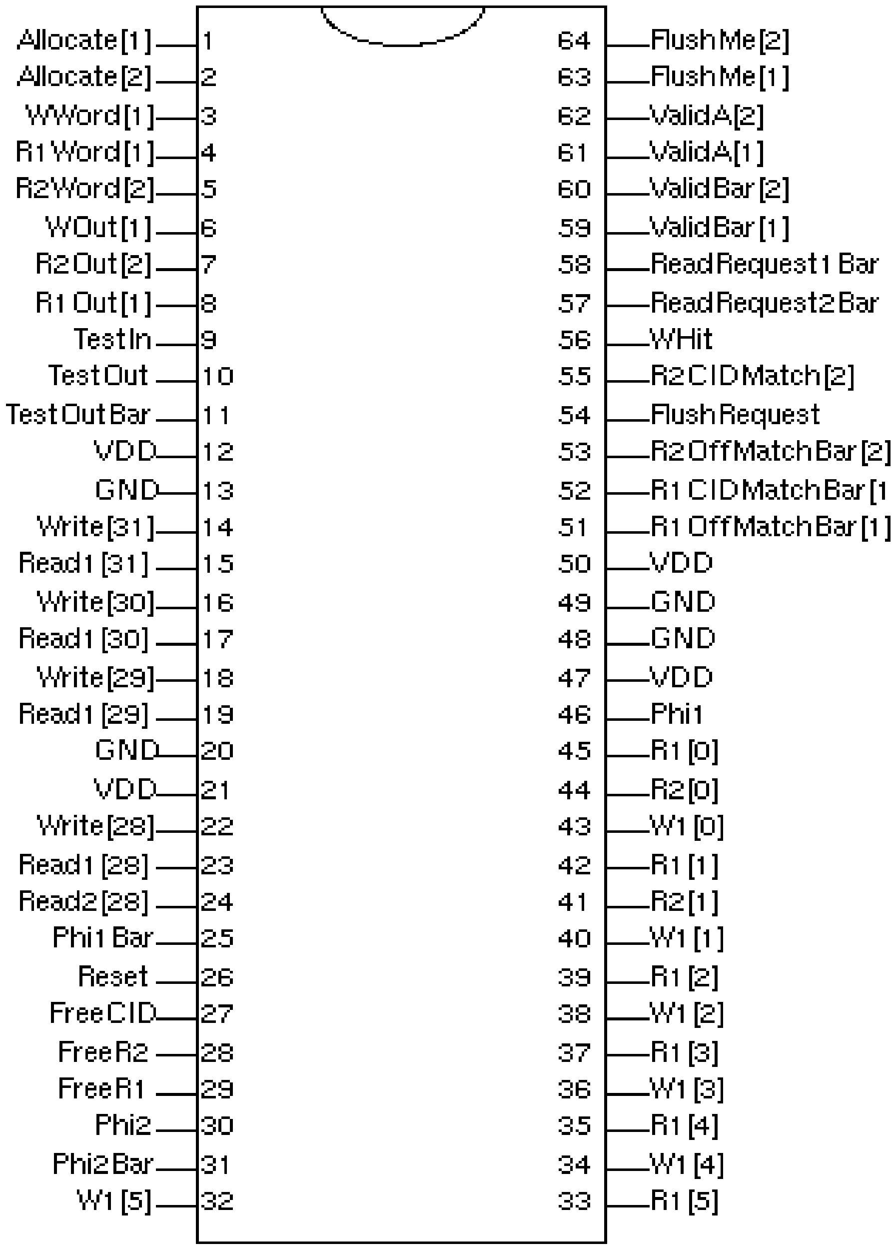


FIGURE A-1. Pinout of the prototype chip.

Figure 1: Context Cache Block Diagram

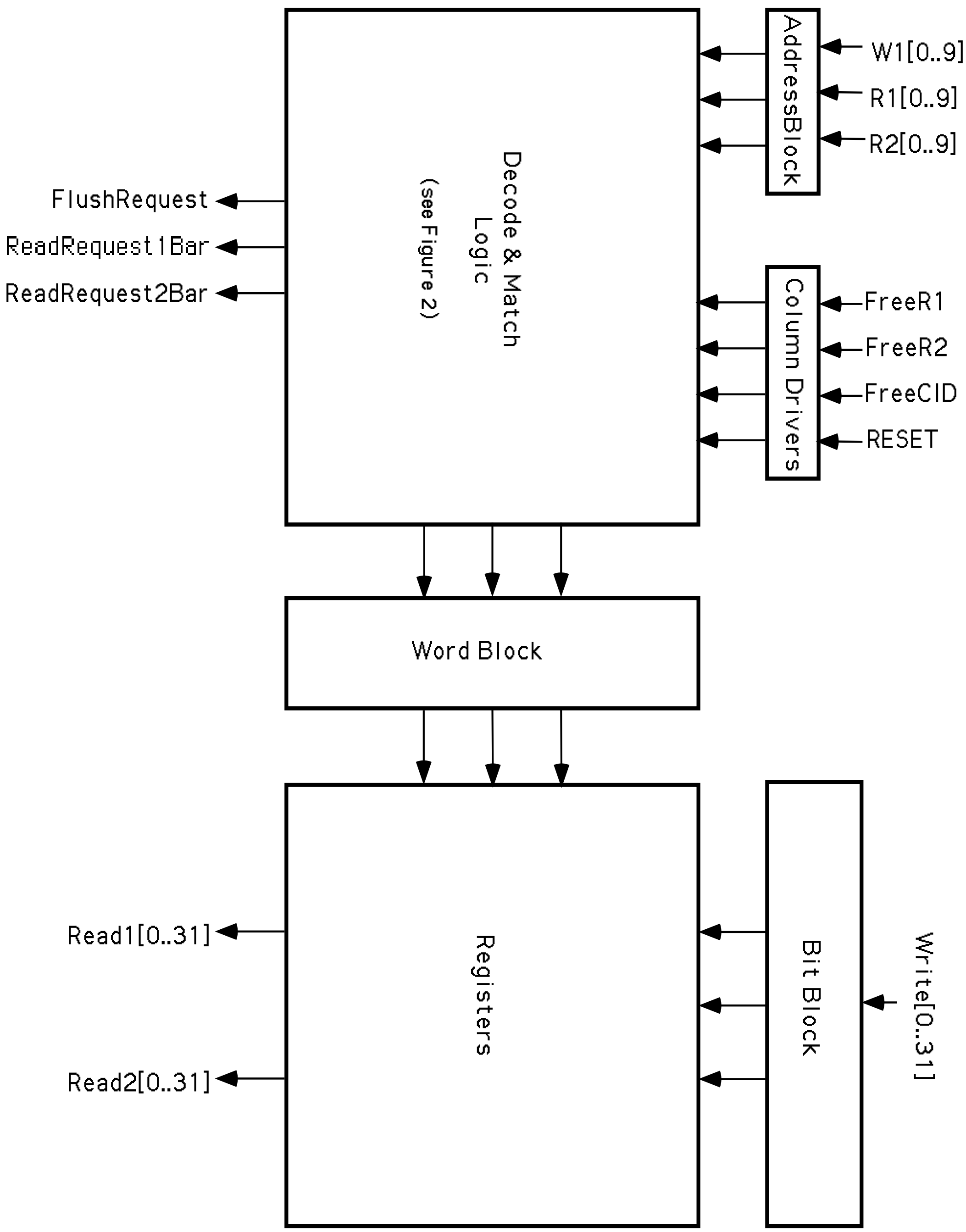


Figure2: Decode/Match Logic

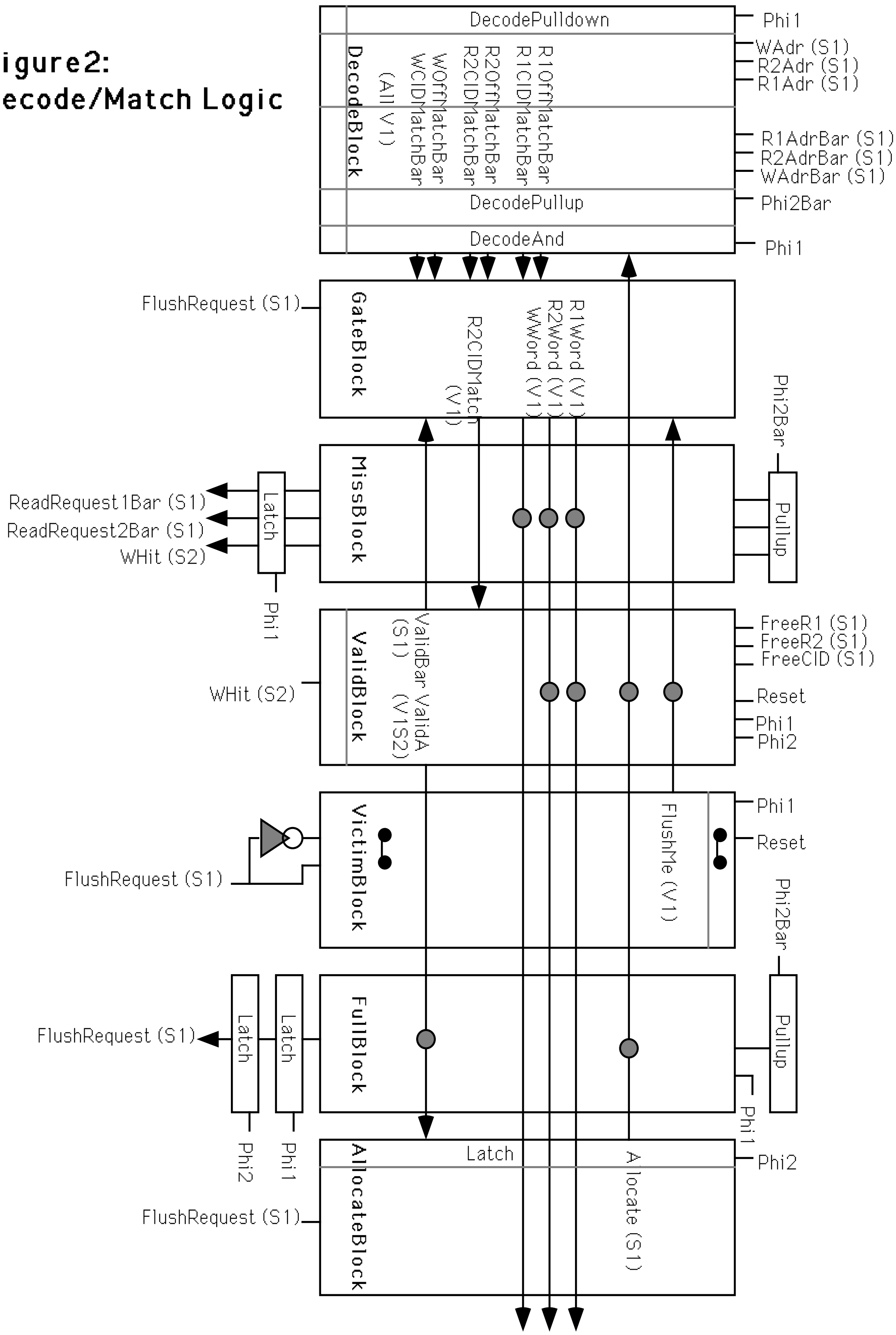
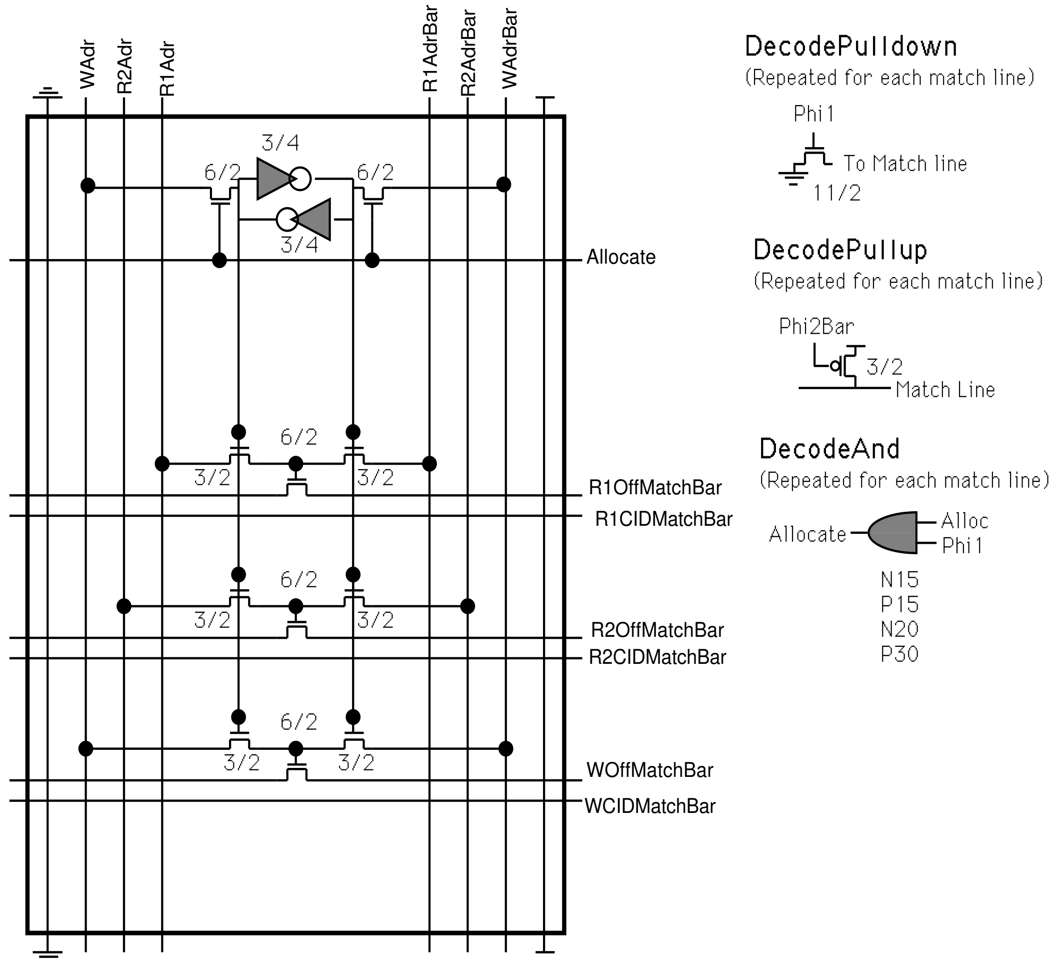
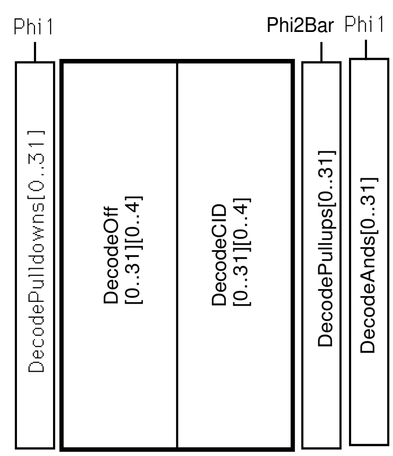


Figure 3: DecodeCell





DecodeCell:

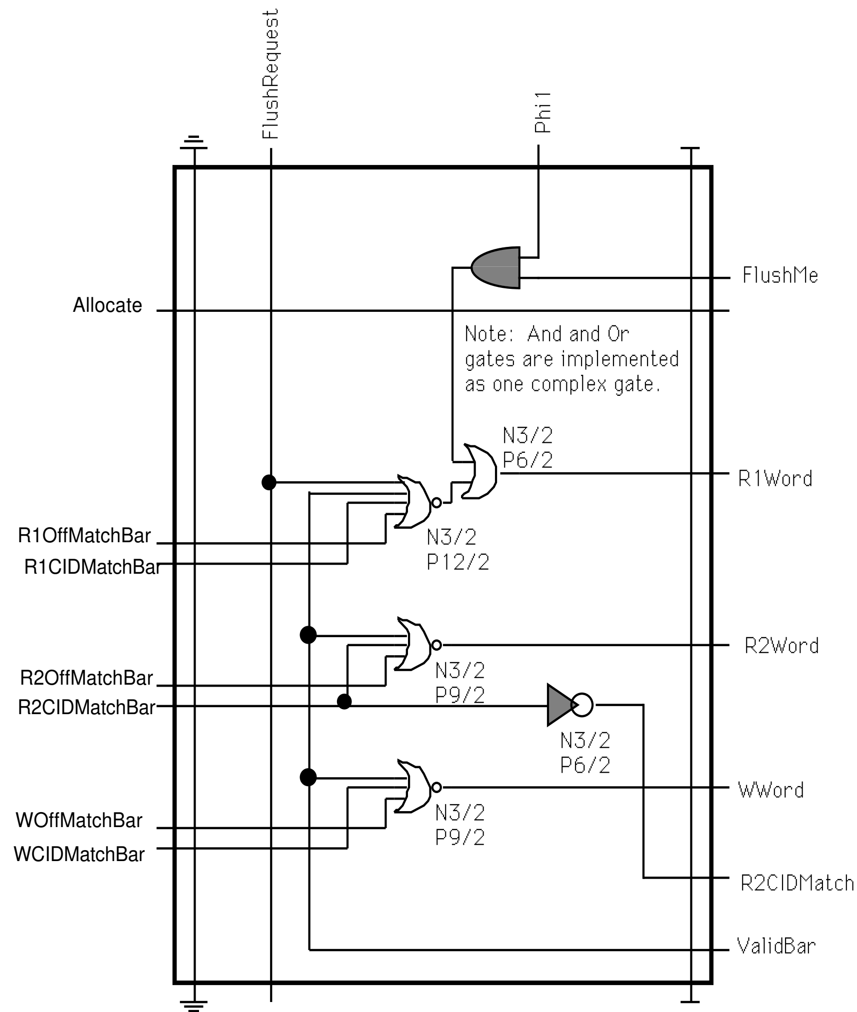
This cell must come in two sexes: DecodeOff and DecodeCID. DecodeOff is shown in this schematic.

There is a danger that, due to charge sharing, the match lines could accidentally be pulled low. Thus, there must be a large amount of capacitance at the pullup on the right side of the block.

A set of And gates gate Allocate with Phi1 to guarantee that the write address is only latched when stable.

DecodeOff[0][*] must have the latches forced to match 0 and DecodeCID[0][*] must be tied low to always match.

Figure 4: GateCell



GateCell:

This cell computes the word line values given the output of the decoders. It also handles a forced read during a victim flush cycle.

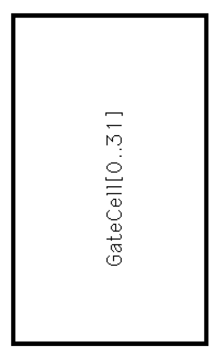
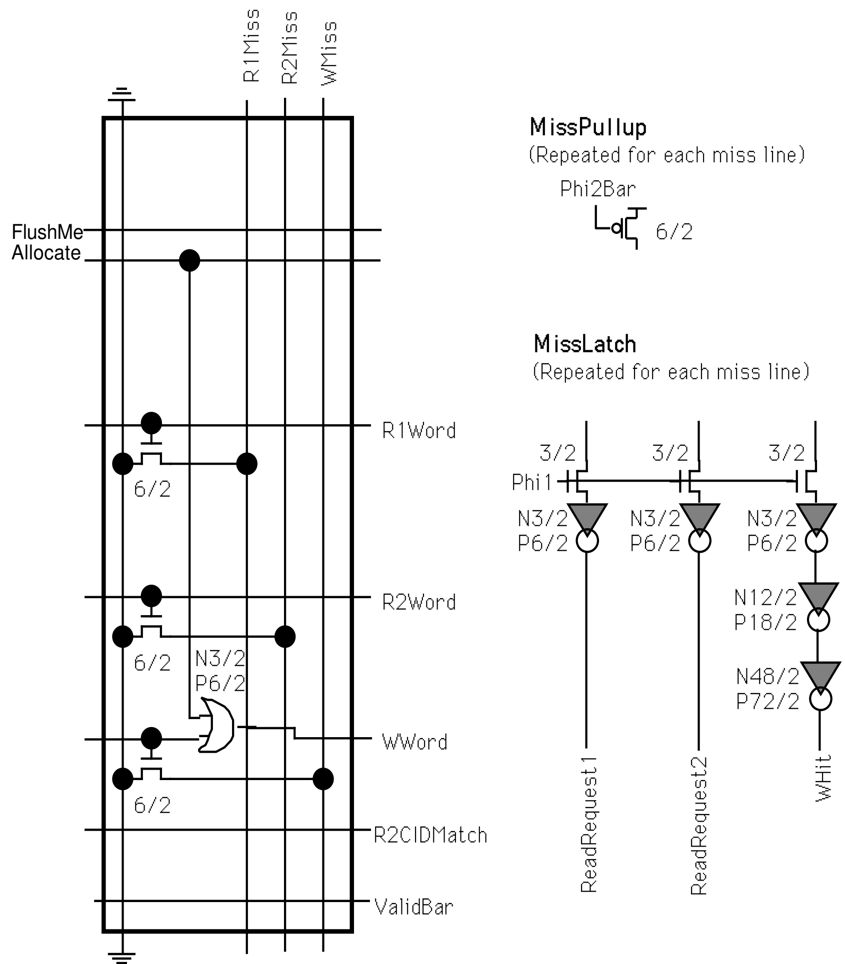


Figure 5:
MissCell



MissCell:

This cell determines if the addresses given to the decoder were missed in the context cache.

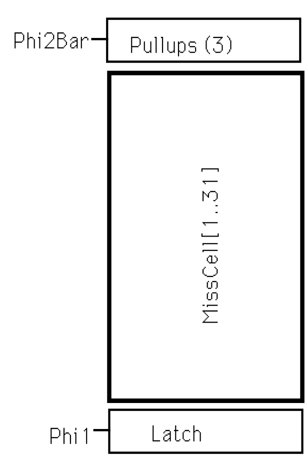
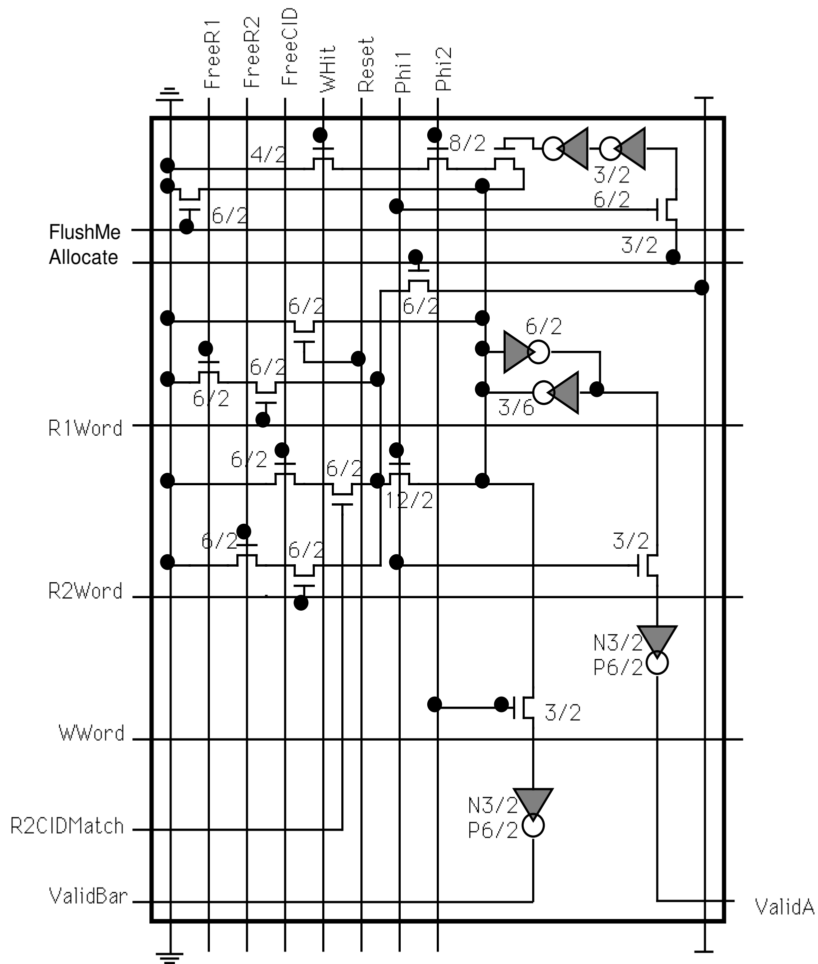


Figure 6: ValidCell



ValidCell:

This cell contains a latch which stores the valid bit and logic to set and clear the latches. ValidF (used to drive FullCell) and ValidBar (used in GateCell) are stable through Phi1. ValidA (used to select the next allocate line) assumes allocates occured and becomes valid late in Phi1.

Note that ValidCell[0] must have the latch tied valid because the zero row is always available.

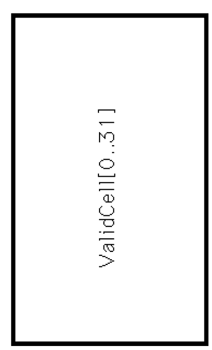
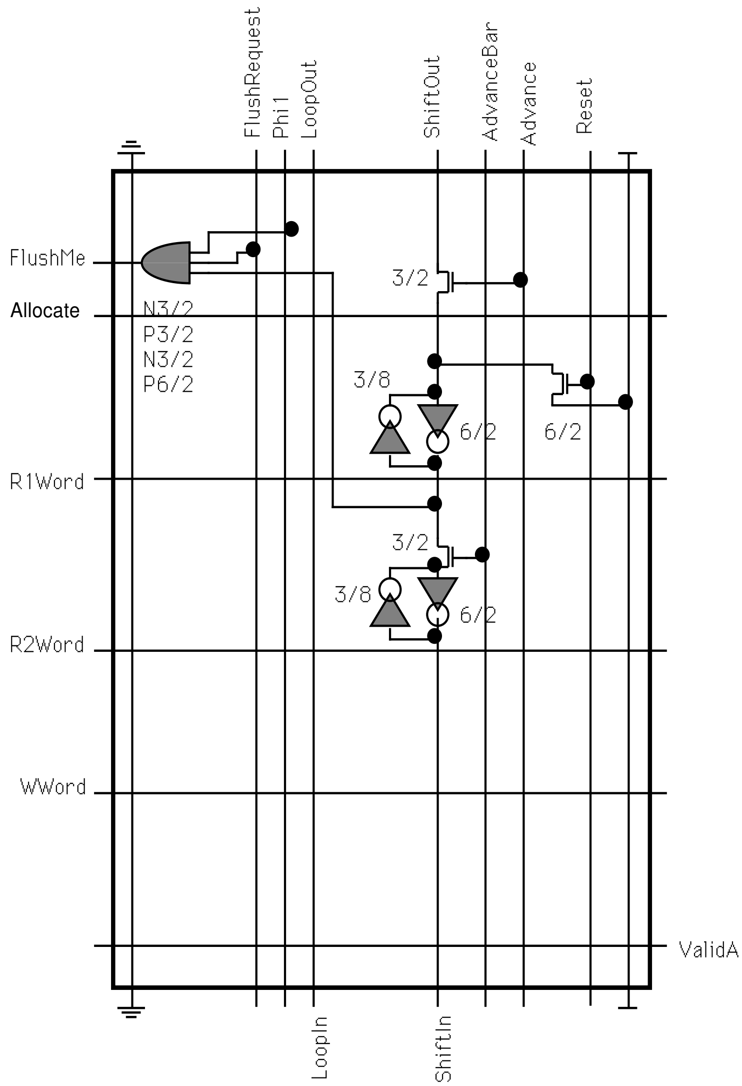


Figure 7:
VictimCell



VictimCell:

This cell is a single stage of a shift register.
The cell comes in two flavors:
VictimCell[1] must load a 0 on reset while the remaining cells must load 1 on reset.
This diagram shows the load 1 flavor.
VictimCell[0] is not in the chain because it should never be flushed.

The shift register advances to choose the next victim when the register file is full.
Note that the latches are drawn upside down; data really propagates up, not down.

Note, also, that there must be a loopback wire from the top to bottom.

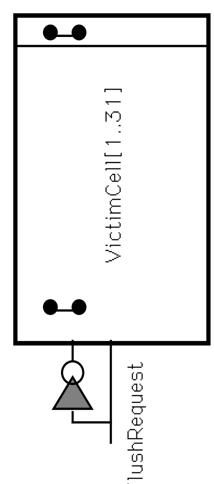
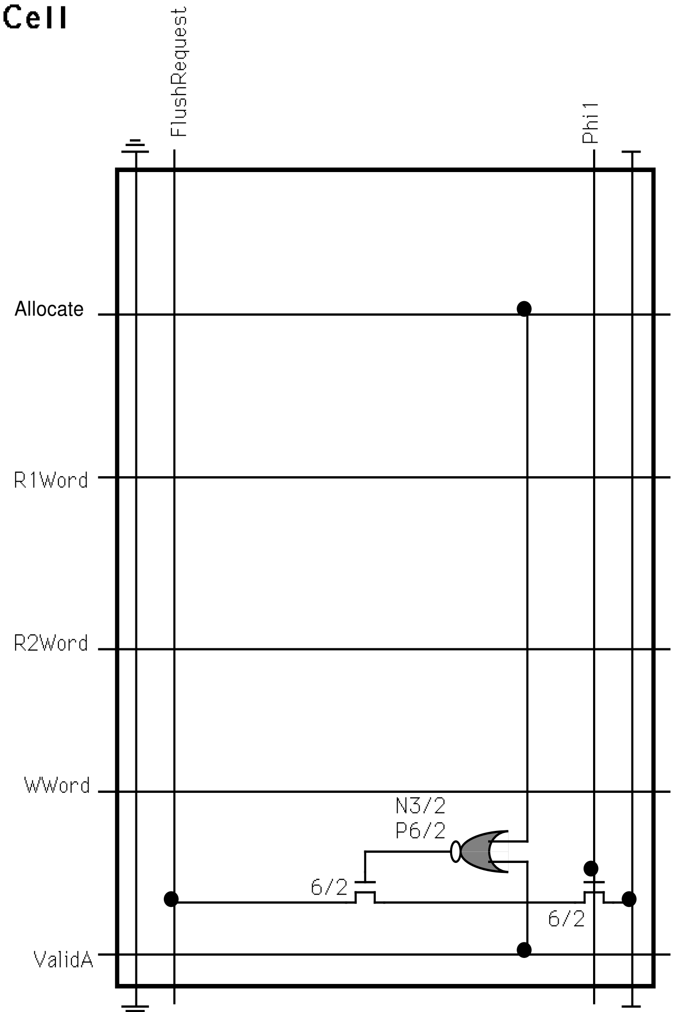
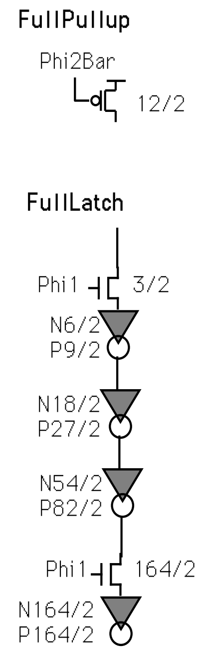


Figure 8:
FullCell





FullCell:

This cell contains pulldowns to set the full line if the last unused line is allocated.

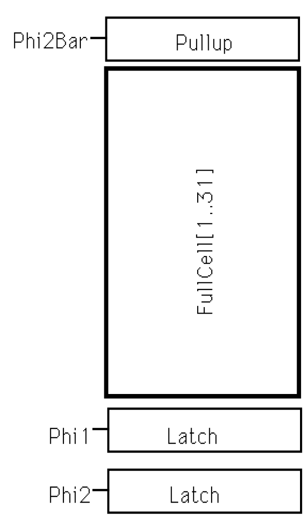
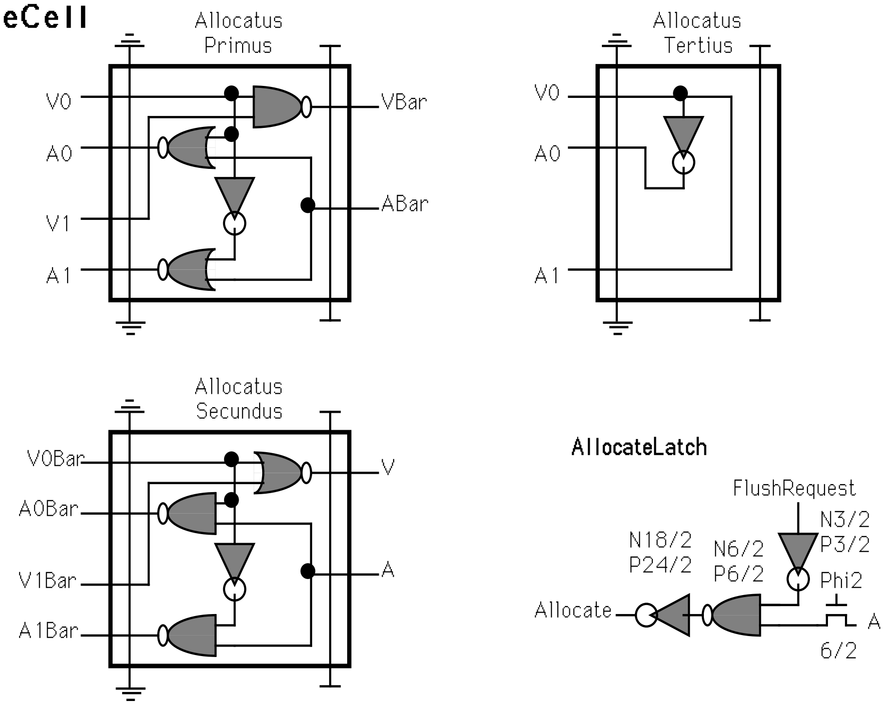


Figure 9: AllocateCell

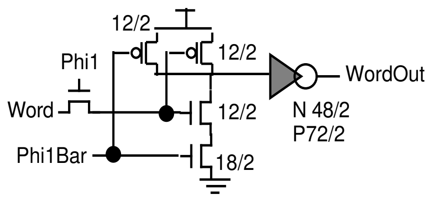


AllocateCell:

This cell consists of tree of three species (classified by Silicon Biologists as Allocatus Primus, Secundus, and Tertius) of subcells that alternate positive and negative logic. The Valid inputs change starting late in Phi1 and the new Allocate line must be latched at the end of Phi2.

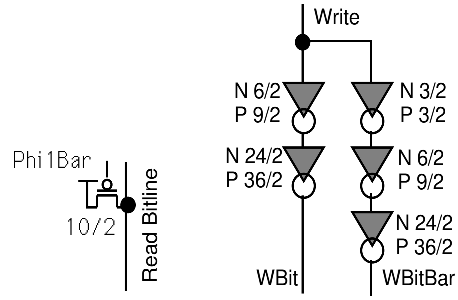
NOT and NOR gates are sized N3/2, P6/2
NAND gates are sized N6/2, P6/2

Figure 10: Drivers & SRAM



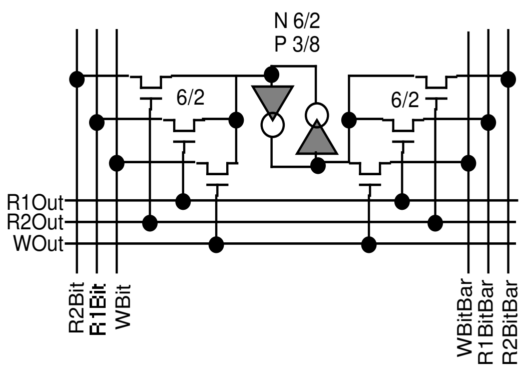
WordDriver

Three word drivers are used on each row in the WordBlock to produce the three word line signals that are driven across the register array.



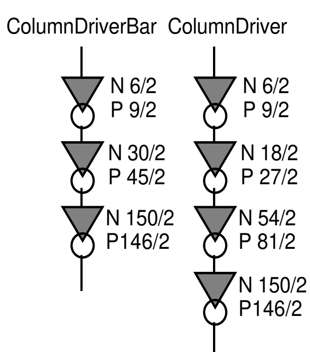
BitDriver

Each column of the register array contains drivers to precharge the read bitlines and drive the write data and its complement down the write bitlines.



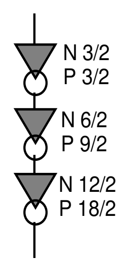
SRAM

The Register block is an array of triple-ported SRAM cells that support two differential read operations and a single write operation each cycle.



Column Drivers

The column drivers are used to drive signals down the tall columns. Four instances of ColumnDriver are used for the three Free lines and the Reset signal. The AddressBlock consists of an array of three instances of ColumnDriver and three instances of ColumnDriverBar above each of the ten Decoder columns. These drivers are used to amplify theread and write address lines.



Buffer

This simple buffer is not used in the ContextCache logic, but is used to drive test outputs along the relatively long lines to the pads.

This page contains schematics for a variety of drivers and the SRAM cell used in the register array.

Bibliography

[1] Anant Agarwal. “Performance tradeoffs in multithreaded processors.” *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] Anant Agarwal et al. “The MIT Alewife machine: A large-scale distributed-memory multiprocessor.” In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[3] Anant Agarwal and Anoop Gupta. “Memory-reference characteristics of multiprocessor applications under MACH.” To appear in SIGMETRICS ’88.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D’Souza, and Mike Parkin. “Sparcle: An evolutionary processor design for large-scale multiprocessors.” *IEEE Micro*, June 1993.

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. “The Tera computer system.” In *International Symposium on Computer Architecture*, pages 1–6. ACM, September 1990.

[6] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks Jr. “Architecture of the IBM System/360.” *IBM Joint Research and Development*, 8(2):87–101, April 1964.

[7] Arvind and David E. Culler. “Dataflow Architectures.” Technical report, MIT/Lab for Computer Science, 1986.

[8] Arvind and Robert A. Iannucci. “Two fundamental issues in multiprocessing.” Technical report, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, May 1987.

[9] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. “I-structures: Data structures for parallel computing.” Computation Structures Group Memo 269, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusettts, February 1987.

[10] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*, chapter The DEC PDP-10, pages 43–45. McGraw-Hill, 1971.

[11] A. D. Berenbaum, D. R. Ditzel, and H. R. McLellan. “Architectural innovations in the CRISP microprocessor.” In *CompCon ’87 Proceedings*, pages 91–95. IEEE, January 1987.

[12] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. “The effect on RISC performance of register set size and structure vs. code generation strategy.” In *International Symposium on Computer Architecture*, pages 330–339. IEEE, May 1991.

[13] Brian Case. “Low-end PA7100LC adds dual integer ALUs.” *Microprocessor Forum*,

pages 9–13, November 1992.

[14] Brian Case. “Sparc V9 adds wealth of new features.” *Microprocessor Report*, 7(2):5–11, February 1993.

[15] David Chaiken. Personal Communication, August 1993.

[16] G. J. Chaitin et al. “Register allocation via graph coloring.” *Computer Languages*, 6(47-57):130, December 1982.

[17] F. C. Chow and J. L. Hennessy. “Register allocation by priority-based coloring.” In *Proceedings of the SIGPLAN ’84 Compiler Construction*, pages 222–232. ACM, 1984.

[18] Paul Chow, editor. *The MIPS-X RISC Microprocessor*. Kluwer Academic Publishers, 1989.

[19] R.W. Cook and M.J. Flynn. “System design of a dynamic microprocessor.” *IEEE Transactions on Computers*, C-19(3), March 1970.

[20] David E. Culler and Arvind. “Resource requirements of dataflow programs.” In *International Symposium on Computer Architecture*, pages 141–150. IEEE, 1988.

[21] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. “Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine.” In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM, April 1991.

[22] William J. Dally et al. “Architecture of a Message-Driven Processor.” In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–205. IEEE, June 1987.

[23] William J. Dally et al. “The Message-Driven Processor: A multicomputer processing node with efficient mechanisms.” *IEEE Micro*, 12(2):23–39, April 1992.

[24] Keith Diefendorff and Michael Allen. “Organization of the Motorola 88110 superscalar RISC microprocessor.” *IEEE Micro*, 12(2):40–63, April 1992.

[25] David R. Ditzel and H. R. McLellan. “Register allocation for free: The C Machine stack cache.” *Proc. Symp. Architectural Support of Programming Languages and Operating Systems*, pages 48–56, March 1982.

[26] Richard J. Eickemeyer and Janak H. Patel. “Performance evaluation of multiple register sets.” In *International Symposium on Computer Architecture*, pages 264–271. ACM, 1987.

[27] Richard J. Eickemeyer and Janak H. Patel. “Performance evaluation of on-chip register and cache organizations.” In *International Symposium on Computer Architecture*, pages 64–72. IEEE, 1988.

[28] James R. Goodman and Wei-Chung Hsu. “On the use of registers vs. cache to

minimize memory traffic." In *13th Annual Symposium on Computer Architecture*, pages 375–383. IEEE, June 1986.

[29] Anoop Gupta and Wolf-Dietrich Weber. "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results." In *Proceedings of 16th Annual Symposium on Computer Architecture*, pages 273–280. IEEE, May 1989. Multithreaded architecture.

[30] Linley Gwennap. "Pentium approaches RISC performance." *Microprocessor Report*, 7(4):2–17, March 1993.

[31] D. Halbert and P. Kessler. "Windows of overlapping register frames." In *CS 292R Final Reports*, pages 82–100. University of California at Berkeley, 1980.

[32] E.A. Hauck and B.A. Dent. "Burroughs' B6500/7500 Stack Mechanism." In *AFIP SJCC*, pages 245–251, 1968.

[33] Raymond A. Heald and John C. Holst. "6ns cycle 256kb cache memory and memory management unit." In *International Solid-State Circuits Conference*, pages 88–89. IEEE, 1993.

[34] John L. Hennessy. "VLSI processor architecture." *IEEE Transactions on Computers*, C-33(12), December 1984.

[35] John L. Hennessy and Patterson David A. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, San Mateo, 1990.

[36] Yasuo Hidaka, Hanpei Koike, and Hidehiko Tanaka. "Multiple threads in cyclic register windows." In *International Symposium on Computer Architecture*, pages 131–142. IEEE, May 1993.

[37] Kirk Holden and Steve McMahan. "Integrated memory management for the MC68030." In *International Conference on Computer Design*, pages 586–589. IEEE, October 1987.

[38] Waldemar Horwat. "Concurrent Smalltalk on the Message-Driven Processor." Master's thesis, MIT, May 1989.

[39] Waldemar Horwat, Andrew Chien, and William J. Dally. "Experience with CST: Programming and implementation." In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 101–109, 1989.

[40] Miquel Huguet and Tomas Lang. "Architectural support for reduced register saving/restoring in single-window register files." *ACM Transactions on Computer Systems*, 9(1):66–97, February 1991.

[41] Robert Iannucci. "Toward a dataflow/von Neumann hybrid architecture." In *International Symposium on Computer Architecture*, pages 131–140. IEEE, 1988.

[42] Robert A. Iannucci. "A dataflow/von Neumann hybrid architecture." Technical

Report TR-418, Massachusetts Institute of Technology Laboratory for Computer Science, May 1988.

[43] Gordon Irlam. *Spa - A SPARC performance analysis package*. gordoni@cs.adelaide.edu.au, Wynn Vale, 5127, Australia, 1.0 edition, October 1991.

[44] Ziv J. and Lempel A. "A universal algorithm for sequential data compression." *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[45] Chris Joerg. Personal Communication, July 1993.

[46] Norman Jouppi. "Cache write policies and performance." Technical Report 91/12, DEC Western Research Labs, Palo Alto, CA, December 1991.

[47] Robert H. Halstead Jr. and Tetsuya Fujita. "MASA: a multithreaded processor architecture for parallel symbolic computing." In *15th Annual Symposium on Computer Architecture*, pages 443–451. IEEE Computer Society, May 1988.

[48] Robert H. Halstead Jr., Eric Mohr, and David A. Kranz. "Lazy task creation: A technique for increasing the granularity of parallel programs." *IEEE Transactions on Parallel and Distributed Systems*, July 1991.

[49] David Keppel. "Register windows and user-space threads on the Sparc." Technical Report 91-08-01, University of Washington, Seattle, WA, August 1991.

[50] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall Inc., 1978.

[51] Tokuzo Kiyohara, Scott Mahlke, William Chen, Roger Bringmann, Richard Hank, Sadun Anik, and Wen mei Huwu. "Register Connection: A new approach to adding registers into instruction set architectures." In *International Symposium on Computer Architecture*, pages 247–256. IEE, May 1993.

[52] David A. Kranz, Robert H. Halstead, and Eric Mohr. "Mul-T: A high-performance Lisp." In *SIGPLAN '89 Symposium on Programming Language Design and Implementation*, pages 1–10, June 1989.

[53] James R. Larus and Paul N. Hilfinger. "Detecting conflicts between structure accesses." In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 22–34. ACM, June 1988.

[54] James Laudon, Anoop Gupta, and Mark Horowitz. "Architectural and implementation tradeoffs in the design of multiple-context processors." Technical Report CSL-TR-92-523, Stanford University, May 1992.

[55] Beng-Hong Lim and Anant Agarwal. "Waiting algorithms for synchronization in large-scale multiprocessors." VLSI Memo 91-632, MIT Lab for Computer Science, Cambridge, MA, February 1992.

[56] D. R. Miller and D. J. Quammen. "Exploiting large register sets." *Microprocessors and Microsystems*, 14(6):333–340, July/August 1990.

[57] Sunil Mirapuri, Micheal Woodacre, and Nader Vasseghi. "The MIPS R4000 processor." *IEEE Micro*, 12(2):10–22, April 1992.

[58] L. W. Nagel. "SPICE2: A computer program to simulate semiconductor circuits." Technical Report ERL-M520, University of California at Berkeley, May 1975.

[59] Rishiur S. Nikhil and Arvind. "Can dataflow subsume von Neumann computing?" In *International Symposium on Computer Architecture*, pages 262–272. ACM, June 1989.

[60] Rishiyur S. Nikhil and Arvind. "Id: A language with implicit parallelism." Technical Report 305, Computation Structures Group, MIT, Cambridge, MA 02139, 1991.

[61] Daniel Nussbaum. Personal Communication, August 1993.

[62] Peter R. Nuth and William J. Dally. "The J-Machine network." In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 420–423. IEEE, October 1992.

[63] Amos R. Omondi. "Design of a high performance instruction pipeline." *Computer Systems Science and Engineering*, 6(1):13–29, January 1991.

[64] John F. Palmer. "The NCUBE family of parallel supercomputers." In *Proc. IEEE International Conference on Computer Design*, page 107. IEEE, 1986.

[65] Gregory M. Papadopoulos and David E. Culler. "Monsoon: an explicit token-store architecture." In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.

[66] David A. Patterson. "Reduced instruction set computers." *Communications of the ACM*, 28(1):8–21, January 1985.

[67] David A. Patterson and Carlo H. Sequin. "RISC I: A reduced instruction set VLSI computer." In *Proceedings of 8th Annual Symposium on Computer Architecture*, pages 443–457. IEEE, May 1981.

[68] G. Radin. "The 801 minicomputer." *Proceedings from the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, March 1982.

[69] Edward Rothberg, Jaswinder Pal Sing, and Anoop Gupta. "Working sets, cache sizes, and node granularity issues for large-scale multiprocessors." In *International Symposium on Computer Architecture*, pages 14–25. IEEE, May 1993.

[70] Gordon Russell and Paul Shaw. "A stack-based register set." University of Strathclyde, Glasgow, May 1993.

[71] Richard M. Russell. "The CRAY-1 computer system." *Communications of the ACM*, 21(1):63–72, January 1978. Author with Cray Research, Inc.

[72] Klaus E. Schauser, David E. Culler, and Thorsten von Eicken. "Compiler-controlled multithreading for lenient parallel languages." UCB/CSD 91/640, University of

California at Berkeley, July 1991.

[73] Charles L. Seitz. “Mosaic C: An experimental fine-grain multicomputer.” INRIA talk in Paris, December 1992.

[74] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. “Splash: Stanford parallel applications for shared-memory.” Technical Report CSL-TR-91-469, Stanford University, April 1991.

[75] Richard L. Sites. “How to use 1000 registers.” In *Caltech Conference on VLSI*, pages 527–532. Caltech Computer Science Dept., 1979.

[76] Burton J. Smith. “Architecture and applications of the HEP multiprocessor computer system.” In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., Aurora, Col, 1981.

[77] V. Soundararajan. “Dribble-Back registers: A technique for latency tolerance in multiprocessors.” BS Thesis MIT EECS, June 1992.

[78] Peter Steenkiste. “Lisp on a reduced-instruction-set processor: Characterization and optimization.” Technical Report CSL-TR-87-324, Stanford University, March 1987.

[79] Sun Microsystems. *The SPARC Architectural Manual*, v8 #800-1399-09 edition, August 1989.

[80] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs. *Computer Structures: Principles and Examples*, chapter Alto: A Personal Computer, pages 549–572. McGraw-Hill, 1982.

[81] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, October 1991.

[82] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Scott, Foresman & Co., Glenview, IL, 1970.

[83] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. “Active Messages: A mechanism for integrated communication and computation.” In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 256–266. IEEE, 1992.

[84] Thorsten von Eicken, Klaus E. Schauser, and David E. Culler. “TL0: An implementation of the TAM threaded abstract machine, version 2.1.” Technical report, University of California at Berkeley, 1991.

[85] Carl A. Waldspurger and William E. Weihl. “Register Relocation: Flexible contexts for multithreading.” In *International Symposium on Computer Architecture*, pages 120–129. IEEE, May 1993.

[86] David W. Wall. “Global register allocation at link time.” In *Proceedings of the ACM SIGPLAN ’86 Symposium on Compiler Construction*, 1986.