A C-ORIENTED REGISTER SET DESIGN

Miquel Huguet

June 1985
CSD-850019

UNIVERSITY OF CALIFORNIA

Los Angeles

A C-Oriented Register Set Design

A thesis submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science

by

Miquel Huguet

Report No. CSD-850019[†]

June 1985

[†]The publication of this report has been supported in part by the Contract 25-3074 from the Sandia National Laboratories

© Copyright by
Miquel Huguet
1985

A la Maria Teresa i al Ricard

To Maria Teresa and Ricard

ACKNOWLEDGEMENTS

I would like to express my gratitude to everybody who has contributed directly or indirectly to the preparation of this thesis:

To my advisor, Professor Tomás Lang, because many of the ideas reflected in this thesis were a result of our discussions and his insistence on finding better solutions. Also for reviewing carefully and with a constructive criticism several previous drafts, and for having served as a model for my research and methodology.

To the other two members of my committee, Professors Miloš Ercegovac and Daniel Berry, because of their suggestions to improve the readability of this text and for their encouragement and support.

To my friends, for helping to create an environment in which research and personal growth can be developed. In particular, I would like to mention Martine Schlag and Pak Chan for their advice and help, Marc Tremblay for implementing the *shift-register file*, Doris Sublette for her assistance in finding several of the references, and Baron Grey for his help with *troff*.

To several people who have provided informative material: Professor Dave Patterson (on RISC), Professor Thomas Gross (on MIPS), Dr. Douglas W. Clark (on VAX-11), Mr. Phil Hermon, Jr. (on C/70), Mr. Bob Lussier (on RIDGE 32), Mr. Bob Ollerton and Mr. Scott Dickson (on CELERITY C1200), and Ms Joan Kerestes (on PYRAMID 90x).

And finally, to my first sponsor, FULBRIGHT/MEC, for enabling me to come to UCLA and supporting me throughout my first two years, and to my current sponsor, CIRIT (*Generalitat de Catalunya*), for supporting the continuation of my research.

ABSTRACT OF THE THESIS

A C-Oriented Register Set Design

by

Miquel Huguet

Master of Science in Computer Science

University of California, Los Angeles, 1985

Tomás Lang, Chair

In this thesis, the architecture of a register file for a processor oriented to execute C programs is considered to study the cost/performance tradeoff. This study is inspired by the same principle which has motivated Reduced Instruction Set Computers, to utilize the processor resources efficiently to support the execution of the most frequent instructions. The design is based on measurements of the execution of a collection of C programs (that include 775 functions and 30,000 lines of code) obtained by modifying the Portable C Compiler. We show that the number of registers required varies and that a large percentage of functions require a small number. Thus, we propose to have a few (2 or 3) window sizes so that the smallest window is always allocated. This window is expanded if the function requires more registers. We use our measurements to determine suitable window sizes and show that for a performance equivalent to the fixed-size window case, the register file can be made significantly smaller. The effects of multiprogramming on the register file are also considered. Moreover, we propose a new implementation for the register file, called the shift-register file, in which the bus size and the access time to the register file is independent of its size and equivalent to the bus size and the access time required for a single-window architecture.

TABLE OF CONTENTS

page

1 INTRODUCTION 1
- 1.1 Antecedents and Motivation 1
- 1.2 Storage Classes for C Variables 4
- 1.3 Static and Dynamic C Measurements 5
- 1.4 Registers versus Cache Memory 7
- 1.5 Register Usage 9
- 1.6 General-Purpose versus Specialized Registers 10
- 1.7 Register Set, Windows, and Function Calls 13
- 1.8 Thesis Organization 15

2 REGISTER USAGE 16
- 2.1 Introduction 16
- 2.2 Local Simple Variables Used by the Programmer 18
  - 2.2.1 Local Simple Variable Usage 18
  - 2.2.2 Automatic versus Register Variable Definitions 20
  - 2.2.3 Number of Local Simple Variables per Function 21
- 2.3 Alias Problem 23
- 2.4 Data Memory Traffic Overhead 26
  - 2.4.1 Data Memory Traffic Caused by Data Objects 27
  - 2.4.2 Overhead for Single Windows 33
- 2.5 Register Allocation Policies 35
- 2.6 Registers for Temporary Results 39
- 2.7 Registers for Optimizing Variables 43
- 2.8 Registers for Parameter Passing 45
- 2.9 Environment Registers 49
  - 2.9.1 Using the Frame Pointer as Argument Pointer 49
  - 2.9.2 Using the Stack Pointer as Frame Pointer 52
  - 2.9.3 Registers Required 53
- 2.10 Registers for Global Simple Variables 53
- 2.11 Conclusions 56

3 MULTIPLE WINDOWS 58
- 3.1 Introduction 58
- 3.2 Data Memory Traffic Overhead for Multiple Windows 60
- 3.3 Fixed-Size Windows versus Variable-Size Windows 62
- 3.4 Multi-Size Windows 63
- 3.5 Overlapped versus Non-Overlapped Windows 67
- 3.6 The Register Set Organization 69
- 3.7 Window Sizes 72
  - 3.7.1 Overlapped Registers 72
  - 3.7.2 Smallest and Largest Window Sizes 73
  - 3.7.3 Unused Registers in Window 74
- 3.8 Register-File Size 77
- 3.9 Exceptions and Context Switching 81
  - 3.9.1 Overhead and Frequency 83
  - 3.9.2 Private Register File 87
  - 3.9.3 Shared Register File 88

3.10 Conclusions 93

4 REGISTER-FILE IMPLEMENTATION 95
4.1 Introduction 95
4.2 Circular-Buffer Register File 95
4.2.1 Window Allocation and Deallocation 98
4.2.2 Overflow and Underflow Detection 99
4.2.3 Context Switching 101
4.2.4 Virtual to Physical Register Number Translation 104
4.2.5 Memory Mapped Registers 106
4.3 Shift-Register File 107
4.4 Overlapped Windows in a Shift-Register File 112
4.5 Multi-Size Windows in a Shift-Register File 116
4.6 Conclusions 119

5 CONCLUSIONS 121

References 123

LIST OF FIGURES

page

Figure 2.1. Example of Alias Problem 23

Figure 2.2. Activation Record Partition with Memory Mapped Registers 26

Figure 2.3. Activation Record Using Frame, Argument, and Stack Pointers 50

Figure 2.4. Activation Record Using Frame and Stack Pointers 51

Figure 2.5. Activation Record Using Only Stack Pointer 53

Figure 3.1. Shared Register File with LIFO order 89

Figure 3.2. Virtual Register Numbers as Mapping Function 91

Figure 4.1. Circular-Buffer Register File 97

Figure 4.2. Register File Overflow 100

Figure 4.3. Mapping with Multi-Size Windows 105

Figure 4.4. Shift-Register File with Fixed-Size Windows 107

Figure 4.5. Saving the Register File on Context Switching 109

Figure 4.6. Cell Connections in the Shift-Register File 111

Figure 4.7. Shift-Register File with Overlapped Windows 112

Figure 4.8. Sharing Four Overlapped Registers 114

Figure 4.9. Shift-Register File with Multi-Size Windows 116

LIST OF TABLES

page

Table 2.1. Local Simple Variable Usage 19

Table 2.2. Local Simple Variables Defined by the Programmer 20

Table 2.3. Register Usage by Benchmark Programmer 21

Table 2.4. Percentage of Defined Local Simple Variables 22

Table 2.5. Data Memory Traffic per Data Object (NROFF) 28

Table 2.6. Data Memory Traffic per Data Object (SORT) 29

Table 2.7. Data Memory Traffic per Data Object (VPCC) 30

Table 2.8. Data Memory Traffic per Data Object (ALL PROGRAMS) 32

Table 2.9. Traffic Caused by Data Objects 32

Table 2.10. Overhead for Single Windows 34

Table 2.11. Effect of Optimization on the VMS C Compiler 37

Table 2.12. Function Parameters 46

Table 2.13. Registers for Parameter Passing 47

Table 2.14. Registers for Global Simple Variables 55

Table 3.1. Overhead for Multiple Windows 61

Table 3.2. Window Size 74

Table 3.3. Unused Registers in Window 75

Table 3.4. A Comparison of Existing Processors 75

Table 3.5. Fixed-Sized Windows 79

Table 3.6. Multi-Size Windows 80

Table 3.7. Measured Exceptions on PDP-11/60 84

Table 3.8. Register File Depth 92

CHAPTER 1
INTRODUCTION

En el pot petit,
hi ha la bona confitura.

(In the small can,
you will find the best marmalade.)

Popular catalan adage

1.1 Antecedents and Motivation

Registers have been used since the beginning of computer design as a mechanism to speed up computations. The first computers had only one accumulator register to perform expression evaluation. When hardware became cheaper, more registers were introduced in the processor: several accumulator registers, index registers, base registers, specialized registers such as the stack pointer, etc. To reduce the complexity in the compiler, registers were made orthogonal, i.e., any operation code can be used with any register [WULF81]. Thus, registers were grouped in a general-purpose register set.

The use of registers reduces the data memory traffic because if operands are already available in processor registers, no memory access has to be generated, and reduces the instruction memory traffic because shorter addresses are used. However, registers have to be saved every time a function is called and restored when the function returns. It has been observed that one of the important factors that influences the design and use of the register file is the overhead produced by the register saving and restoring [LUND77, PATT82].

Recent advances in compiler technology, specially in register allocation optimization [CHAI81, CHAI82, ANKL82, CHOW84, MCKU84], have resulted in a better usage of the register set. Thus, the compiler can allocate more simple variables to registers, and if the variables have disjoint lifetimes, they can share the same register without register saving and restoring. To avoid the development cost of an optimizing compiler for every language and every machine [JOHN81], optimizing compiler research pursues independence of the source language and

the target machine to increase their portability [AUSL82, TANE83].

Furthermore, advances in hardware technology that make possible a large number of registers in a single chip [SITE79] have made possible the reduction of register saving and restoring overhead. To obtain this, a new bank or window of registers is made active on each function call [SITE79, DANN79, LAMP82] so that registers have to be saved only when no more windows are available in the register file. Patterson and Séquin [PATT82] have already quantified the reduction in the data memory traffic due to the register saving and restoring overhead for two benchmark programs comparing a multiple-window architecture (RISC) with a single-window architecture (VAX-11): from 444K to 8K for the puzzle benchmark, and from 696K to 4K for the quicksort benchmark.

Both compiler and hardware technological advances have caused the proliferation of new register-oriented processors either with a VLSI design: MIPS [HENN82], and RISC I and II [PATT82, KATE83]; or with a MSI/LSI multi-chip design: IBM 801 [RADI82], RIDGE 32 [BASA83], PYRAMID 90x [PTC83], and CELERITY C1200 [OLLE85]. Some of these processors have introduced a new instruction set design style, called *Reduced Instruction Set Computers* as a contrast to the tendency of increasing the complexity of the processors.

At this time there is no clear understanding of the elements that define a Reduced-Instruction-Set-Computer approach. Patterson [PATT85] has characterized the existing Reduced Instruction Set Computers by the following factors: operations are only register-to-register, memory operands must be loaded first to a register to be operated, instruction decoding is simpler, operations and addressing modes are reduced, and branches avoid pipeline penalties. The basic idea is to execute fast the small number of instructions that have been shown to be very frequent [ALEX75, SHUS78, SWEE82, MCDA82, WIEC82, CLAR82]. In this same direction the availability of a multiple-window register file has been proposed as a method to reduce the large overhead in function calls [PATT82].

The design of a register set was studied by Lunde [LUND77]. He measured several algorithms programmed in different languages (ALGOL, BASIC, FORTRAN, and MACRO) and concluded that the register set should include: two floating-point accumulators, two fixed-point accumulators, and eight registers with simple fixed-point operations. However, modern processors are constantly increasing their register sets: 16 registers for the RIDGE 32; 32 for the RISC, MIPS, and IBM 801; and 64 for PYRAMID 90x and CELERITY C1200.

Furthermore, when a multiple-window register file is also available in the processor to reduce the function call overhead, the total number of processor registers is even larger: 138 for RISC, 528 for PYRAMID 90x, and 4096 for CELERITY C1200. While this reduces the overhead in function calls and returns, it might have the following drawbacks: it uses a large chip area in VLSI implemen-

tations or a large number of chips in a MSI/LSI implementation, it increases the amount of processor context to be saved on context switching, and it increases the processor cycle-time due to the long data busses [HENN84].

Therefore, the basic idea given by Reduced Instruction Set Computers, to utilize the resources efficiently to support the execution of the most frequent instructions, has not been applied in the design of the processor register set. Thus, the main motivation of this thesis is to balance the cost/performance tradeoff of the register file so that the most frequently executed functions have enough registers to be executed and so that the least amount of hardware resources (registers) is unutilized.

To study this cost/performance tradeoff, we consider the architecture of a register file for a processor oriented to execute programs written in C [KERN78]. C has been selected because it is widely used for system programming [FEUE82], it has a type flexibility that allows to program system functions traditionally coded in assembler [KERN81, POST83], it has been used to implement UNIX*, it can be used as an intermediate language for other block-structured languages such as ADA** [HILL83], and has been used to implement other languages such as LISP and PROLOG. No attempt is made in this thesis to generalize the measurements obtained for the C programs. The reader is referred to [WEIC84] or [HUCK83] for a comparison of the characteristics of different programming languages to C.

To study the design of a C-oriented register set, first we consider the use of the registers made by the compiler to store different types of information (variables defined by the programmer, variables defined by the compiler itself, process-execution environment information, etc.), the problems that the compiler has to store this information in registers, and the number of registers required to store this information (Chapter 2). Second, we discuss how the register set is organized with a multiple window approach to balance the cost/performance tradeoff (Chapter 3). Finally, we study how the register file can be implemented to reduce the drawbacks in size and speed (Chapter 4).

We show that the number of registers required varies and that a large percentage of functions require a small number. Thus, we propose to have a few (2 or 3) window sizes so that the smallest window is always allocated. This window is expanded if the function requires more registers. We use our measurements to determine suitable window sizes and show that for an equivalent performance to the fixed-size window case the register file can be made significantly smaller. The effects of multiprogramming on the register file are also considered and we show that either two private register files or a shared register file among the running

*UNIX is a trademark of Bell Laboratories

**ADA is a trademark of the U.S. Department of Defense (Ada Joint Program Office)

user process and the exception handlers is required. Moreover, we propose a new implementation for the register file, called the shift-register file. The shift-register file makes the bus size and the access time to the register file independent of its size, and equivalent to the bus size and the access time required for a single-window architecture.

This first chapter is organized as follows. Section 1.2 reviews some terminology used to define storage classes for C variables. Section 1.3 introduces the static and dynamic measurements gathered to justify the design of a C-oriented register set. Section 1.4 comments the advantages of having local simple variables in registers instead of having them in a memory stack with a cache. Section 1.5 discusses the register usage, i.e., the type of information that the programmer (compiler) stores in registers. Section 1.6 compares two different organizations for the register set (general-purpose registers and specialized registers). Section 1.7 considers the influence of function calls and returns on the design and use of the register file. Finally, Section 1.8 presents the organization of this thesis.

1.2 Storage Classes for C Variables

Each variable in C belongs to one of the following four storage classes: external, static, automatic, or register. The storage class of a variable determines its lifetime and the memory segment where it is stored (data segment for external and static variables, and stack segment or processor registers for automatic and register variables).

External variables exist throughout the execution of the entire process. *Static variables* also exist throughout the execution of the entire process, but they can be referenced only by functions defined in the same module, i.e., they are only known in the same module where they are defined, the compiler does not allow functions outside of this module to refer to them. Static variables are useful when a function needs to remember a value from call to call; for example, for a function to generate random numbers. Storage for external and static variables is allocated statically during compilation time. Let us refer to both classes as global variables. Observe that the language does not require any special protection for static variables. Both external and static variables share the same data segment and, therefore, can use the same addressing mechanism to be referenced.

Automatic variables are local to each block, that is, to each set of statements enclosed between brackets ({,}). Space for automatic variables is dynamically allocated when the block is entered and deallocated upon exit. To avoid the cost of having to allocate/deallocate space per block basis, automatic variables are allocated when a function is called and deallocated when the function returns; however, the compiler does not allow statements outside a block to refer to local variables defined inside the block. The allocated space is called the activation record of the function; it includes not only automatic variables, but also parameters received by the function and some environment information (return address,

dynamic link, etc.).

Register variables are also local to each block. They are used as a hint to the compiler to store them in the processor registers. If there are more register definitions than processor registers, then the ones which cannot be stored in registers are treated as automatic variables. However, an optimizing compiler can share the same processor register among several variables with disjoint lifetimes [CHAI81]. Three different register allocation policies are discussed later in Section 2.5. Only simple variables can be defined of type register.

1.3 Static and Dynamic C Measurements

One of the goals of this project has been to gather the necessary information to justify the design of a C-oriented register set. Static measurements are useful to justify issues related to code size and dynamic measurements to justify issues related to execution time. Both measurements were obtained modifying the Portable C Compiler [JOHN79, LION79]. Four programs were measured:

1) A program that plots VLSI mask layouts on a dot plotter (CIFPLOT), which includes 274 functions in 9750 lines of code.

2) A text formatting program (NROFF), which includes 226 functions in 6500 lines of code.

3) The UNIX sorting program (SORT), which includes 21 functions in 916 lines of code.

4) The Portable C Compiler for the VAX-11 (VPCC), which includes 252 functions in 12,800 lines of code.

However, only static measurements are available for the CIFPLOT program, because there is no driver for the VLSI laboratory plotter installed in our system. These programs have been selected because they have already been used at Berkeley for the RISC design [PATT82]. At Berkeley, they have also used different short benchmark programs to measure the performance of RISC with respect to other machines [PIEP81, PATT82a]. We have also measured the same benchmark programs, but we do not use them to justify the design of the C-oriented register set. The reason is that they are too short to be representative enough when we want to determine, for instance, the stack depth of the executed programs, or the number of local simple variables defined by the programmer for each function. This does not imply that the short benchmark programs cannot be used to compare execution times for different architectures, although the purpose of the benchmark (or what the benchmark is measuring) has to be kept in mind to interpret the results.

The measurements were made using the Portable C Compiler generating code for VAX-11. For the static measurements, only the first pass of the Portable C Compiler is used. In this case, counters are kept to accumulate local simple variable definitions and data object usage as soon as they have been parsed. For the dynamic measurements, code was generated so that every time that a specific event is executed (for instance, a local simple variable is referenced) a counter associated to such event is incremented. Program execution was intercepted at the end to print the contents of the counters.

The measurements are used in this thesis to deduce the number of registers required by C programs and to compute the data memory traffic overhead caused by register saving and restoring. Some more measurements were taken, but they are not shown in this thesis because they are not relevant to our discussion. The measurements are not reported in this section: they are introduced in the sections where needed. The following have been measured:

(a) The number of local simple variables defined and used by the programmer for each function (Section 2.2).

(b) The number of data objects executed to determine the data memory traffic caused by each data object and the register saving and restoring overhead (Sections 2.4 and 3.2). Data objects are classified according to their storage class and their type (local simple variables, arrays, pointers to simple variables, pointers to structures, etc). The purpose of this classification is to allow us to compute for each executed data object the number of memory references generated (Section 2.4.1).

(c) The number of parameters passed to a function and the type of expressions to evaluate for parameter passing (Section 2.8).

(d) Functions that require the creation of an activation records in the stack because no more registers are available, and the number of registers unused in the window for each function (Section 3.7).

(e) Overflows generated by different window and register file sizes (Section 3.8).

(f) Distribution of the stack depth in the register file, i.e., number of registers present in the register file per function (Section 3.9.3).

1.4 Registers versus Cache Memory

The current activation record can be stored either in a memory stack or partially in a memory stack and partially in registers as discussed in the next section. There are three major motivations for having as many variables as possible in registers:

1) Instruction formats require fewer bits to specify the address of an operand in a register than one in memory.

2) Access to variables is faster from registers than from memory, although a cache also provides fast access to operands.

3) There is less memory traffic because some operands are already in the processor registers.

Both cache memory and registers are valid methods to get fast access to operands for local simple variables. However,

(1) Cache memories cause some overhead in checking if the data is available or not. Furthermore, if the data is not available, the processor will have to wait for a new block from main memory.

(2) Cache memories do not reduce the instruction length because a full address specification is still required to refer to the operand. The address of a local variable stored in the activation record is usually specified as a displacement relative to the frame pointer. Fewer bits are required to specify a register address than to specify the displacement.

(3) The addressing mode specifier has to be decoded and the operand address has to be computed every time the instruction is executed.

Of the three mentioned problems the most critical is the third one. Every time that the variable has to be fetched the contents of the frame pointer register and the displacement have to be added so that even if the variable has been used recently and it is in the cache, an extra cycle is required to perform the addition. If the local variable were stored in a register, then not only fewer bits would be required, but also no overhead for address computation would be incurred.

To evaluate this overhead we can consider, for instance, measurements done by Norton and Abraham [NORT83] on a modification to the VAX-11 implementation. They design an instruction buffer such that any reference to a local variable is resolved (i.e., the offset is added to the frame pointer to compute the variable's virtual memory address) and its virtual address is stored in the instruction buffer, rather than its addressing mode specifier (offset and frame pointer). While this instruction is in the buffer an execution cycle is saved because the vir-

tual address has not to be computed again. The new implementation was measured using two programs: the UNIX file comparison program *diff* and an eight by eight matrix multiply program. The improvement for the first program ranged from a factor of 1.76 with an I-cache size of 64 instructions to a factor of 2.11 with an I-cache of 256 instructions. The improvement for the second one ranged form a factor of 2.65 with an I-cache of 64 instructions to a factor of 2.75 with an I-cache of 256.

Therefore, although both cache and registers fulfill one of the three major motivations for having data in registers (fast access to operands), registers are more efficient than cache to store local simple variables because:

(1) The complexity of computing the local simple variable address is only paid once during compilation because the compiler allocates the variable to a register and its address does not have to be computed every time it is referenced.

(2) The instruction size is reduced because fewer bits are required to refer to a local simple variable stored in a register.

(3) Only one copy of the local simple variables is available so that no multiple updates in different hierarchical memory levels have to be made.

The drawback is that an optimizing compiler is required to perform register allocation and to use the register set efficiently. Not everyone agrees in moving towards this direction: there already exists a proposal to simplify compiler complexity moving it to the architecture [HARB82], and some authors [JOHN81] claim that optimizing compilers should be eliminated. Moreover, having local simple variables allocated to registers increases register saving and restoring memory traffic overhead because registers have to be saved when a function is called, and restored when the function returns. However, in Section 2.4 we will see that the register saving and restoring memory traffic is balanced with the reduction in data memory traffic caused by the data objects when local simple variables are allocated to registers. Therefore, registers should be used to store local simple variables.

Cache memories also have a similar problem to registers. Since the cache memory has a copy of the data object, the original value of the data object has also to be updated when the copy is modified. No modification is required while the data object is only being read. Our measurements have shown that, on the average, for every executed data object which is written, 7.3 data objects are accessed (including the one being written). The original value in memory can be updated at the same time that the copy is updated (write-through) or when the copy has to be replaced (copy-back) [SMIT82]. No comparison is made in this thesis about the data memory traffic generated for both cases (cache and registers). The selection of registers is based on the three reasons given above and on the reduction on register saving and restoring overhead that we obtain using mul-

tiple windows shown in Section 3.2.

Although local simple variables are allocated to registers rather than to memory, this does not imply that cache memories should be discarded. They are necessary to keep instructions and data that cannot be allocated in registers. Moreover, current research on memory caches is showing that it is better to separate caches for data and for instructions because they require different organizations (a instruction cache is simpler because it is read only), different algorithms for pre-fetching, and different algorithms for replacements [SMIT85].

Besides these two approaches (registers and cache), a third approach has also been proposed: to store the activation record in Writable Control Store (WCS) [WAKE83]; however, no study was made about its implementation. This approach is not discussed in this thesis because we are focusing our design towards a VLSI hardware implementation, not a microprogrammable one.

1.5 Register Usage

Registers cannot be used to store non-simple variables (arrays, structures, or unions) because of their size. External and static non-simple variables must be stored in the data segment and automatic non-simple variables in the memory stack.

Registers in the general-purpose register set cannot be used to store global (external or static) simple variables either because, in C, functions can be compiled separately so that the compiler does not know where global variables have been allocated when the other modules were compiled. However, an alternative solution exists to have global simple variables in a special set of registers referenced with a memory address instead of a register address. This is discussed in Section 2.10.

Consequently, registers can be used for the following purposes:

(1) To store local (automatic or register) simple variables defined by the programmer. Local simple variables correspond not only to data of a fundamental data type (integer, character, unsigned, ...), but also to pointers to variables of any type, i.e., any variable that can be stored in a register.

(2) To perform expression evaluation, i.e., to store temporary results.

(3) To store some values, basically addresses, generated by the compiler to optimize program execution. For instance, if a multi-dimensional or unidimensional array is referenced inside a loop, then the loop invariant address computation can be removed from inside the loop and stored in a temporary variable so that it is only computed once.

(4) To store the run-time process environment state, i.e., the address of the next instruction to execute, the address of the top element in the execution stack, the address of the last activation record in the stack, etc.

(5) To pass parameters to the callee and to return a value back to the caller. C functions cannot pass or return arrays, structures, unions, or functions; they can only pass or return data of a fundamental data type or pointers. So, parameters and the return value fit in registers.

(6) To store *bases* to compute the effective address of operands as *base + displacement.*

(7) To store *indices* to compute the effective address of operands as *base address + shifted-index.* The index register is shifted depending on the number of bytes of the operand. If the base address register not only contains the base address, but also a bound value used to check during run-time execution that the shifted-index displacement is smaller than the bound, then the base address register is called a *descriptor.*

In the next chapter we present measurements to determine the number of registers that can be used effectively for each of these purposes. Section 2.2 presents how many local simple variables are defined by the programmer for each function. Sections 2.6 and 2.7 discuss how many registers are used to store temporary results and optimizing variables. Section 2.8 studies different alternatives for parameter passing and determines how many registers are required to pass parameters through them. Section 2.9 shows how many registers are required to store the run-time process environment. The use of special registers such as *bases, indices,* or *descriptors* more related to addressing modes is not discussed in this thesis.

1.6 General-Purpose versus Specialized Registers

Two possible organizations are possible for the register set: to have general-purpose registers (for all uses) or to have specialized registers. The organization of the register set has a direct influence on both the instruction-set format and the register-set implementation.

Regarding the instruction-set format specialized registers have two main advantages: less bits are required to refer to a specialized register and they are better protected. Specialized registers are usually referenced implicitly by the operation code so that the instruction format requires less bits. For example, the stack pointer register in zero-address architectures, the accumulator register in one-address architectures, and the program counter register in branch and function call instructions.

Also specialized registers offer better protection, specially for environment registers. If they can only be modified by special instructions (i.e., the program counter modified only by branch and function call instructions, the frame pointer only by function call instructions, etc.), then the programmer cannot change any of them on purpose or by mistake. For example, let us consider the PDP-11 [DEC78]. The program counter and the stack pointer are available as general-purpose registers. If a programmer is using the stack pointer register to store some characters into the stack and an exception occurs, then we can have a program trap error depending on the value of the stack pointer. If it is even, then the exception function will push the program counter and the stack pointer into the stack and no error will be generated. If it is odd, then an odd address program trap error will be generated because it is not possible to move a word to an odd address. This error is caused not only by exceptions, but also by the function call instruction.

Although VAX-11 [DEC79] has also the stack pointer as a general-purpose register, this problem is solved: when an exception occurs or a function call instruction is executed, then the two lower bits of the stack pointer are forced to zero and the old contents are saved into the stack so that it can be restored accordingly.

In spite of both advantages (instruction length and protection), architectures tend to use more general-purpose registers because:

1) General-purpose registers allow more orthogonality in the instruction set: one op-code is required to perform a specific operation with any of the registers.

2) General-purpose registers simplify the task of the compiler writers because they have less special cases to consider in code generation [WULF81].

3) Environment registers in a general-purpose register set are protected by the compiler. Programmers will be using high level languages and they will not have access to these registers.

Specialized and general-purpose registers can (and sometimes must as we will see below) be combined to get the advantages of both. For example, the MC68000 [MC79] has sixteen 32-bit general-purpose registers divided into two sets, eight data registers and eight address registers so that the address mode decides which register set is going to be used and the instruction format only requires three bits to specify the register address. The MIPS machine [GROS83] has 16 general-purpose registers and 6 special registers; these registers are referenced implicitly by certain operation codes; for instance, there is a 16-bit shift format (eight bits for the operation code, four bits for the source register, and four bits for the destination register) that uses one of the special registers to indicate the shifting amount.

Yuen [YUEN84] has proposed a new approach to try to shorten the number of bits required to specify an operation: to use implicit registers that do not depend on the operation code, but on the context left by the previous executed instruction. For instance, one instruction refers a register as a general-purpose register and the following ones can use this register without explicitly referring it in the instruction format. The drawback of this approach is that new operation codes must be introduced ones to work with an explicit register and others to work with an implicit register. In a later paper, Hor and Yuen [HOR85] present an instruction set with these characteristics.

The implementation of the register set is affected by its organization. With general-purpose registers the implementation is limited to perform read/write operations on the register set; however, with special registers the instruction execution can make better use of the parallelism so that different actions can occur in parallel. For instance, let us consider the current-window pointer and the first-window pointer required to manipulate a register file with multiple windows organized as a circular buffer [KATE83]. The current-window pointer points to the current buffer (window) and the first-window pointer points to the first buffer (window) in the file. The current-window pointer must be used to translate any virtual register number and both must be used to detect if a memory address refers to a physical register location if registers are memory mapped (see Section 4.2.5). If they were in the general-purpose register set (for example, as global registers and implemented in a register file for global registers), then both registers would have to be read to perform these operations before reading the operand; as a consequence the time required to execute an instruction would increase. Therefore, both registers must be specialized register so that they can be operated in parallel with the ALU execution. This does not imply that for the users' point of view specialized registers cannot be referenced as part of their register set.

It is also better to have the program counter (PC) as an specialized register. For example, let us consider four computers, two designed at Digital Equipment Corporation and two designed at the University of California at Berkeley: PDP-11/60, VAX-11/750, RISC II, and SOAR.

In PDP-11/60 [DEC77] there is not automatic instruction prefetch; only register-to-register instructions perform automatic prefetch to save execution cycles. The PC is a general-purpose register and it is implemented in the register file together with the other registers. It is incremented explicitly by the microinstruction that performs the fetching of the next instruction. The ALU and the register file are idle when a memory operation is required so that the cycle is used to increment the PC.

In the VAX-11/750 implementation [DEC80] the program counter is implemented as a specialized register although for the user it is a general-purpose register. The instruction stream in VAX-11/750 is prefetched automatically, four bytes at a time, and loaded in an 8-byte instruction buffer; the program counter is incremented automatically at the same time that a new word is brought into the instruction buffer. So, no explicit micro-instruction has to be given to increment the program counter.

Katevenis [KATE83] proposes to separate fetching and execution units in the RISC II architecture. The program counter is moved to the instruction fetch-and-sequence unit so that control-transfer instructions are transparent to the execution unit. The main advantage of this implementation is that if a two-port instruction cache is available, then unconditional branches can be executed with zero-delay time. This scheme is only possible if the program counter is not a general-purpose register and the execution unit does not refer implicitly to the program counter in any instruction. The program counter in RISC II is not a general-purpose register, but the jump instructions use it implicitly as a PC-relative addressing mode; therefore, RISC II implementation makes no use of this scheme even though a two-port instruction cache is available.

However, in SOAR [UNGA84] jump instructions contain the absolute address of the target instruction so that no address computation is required and the instruction prefetch penalty for jumps is eliminated.

In conclusion, general-purpose registers should be used to store local simple variables defined either by the programmer or by the compiler. Specialized registers can be used to implement environment registers (at least, the current-window pointer and the first-window pointer in multiple-window architectures), although they can also be referenced using a general-purpose register. This discussion finishes in Section 3.6 where it is presented how the C-oriented register set is organized.

1.7 Register Set, Windows, and Function Calls

In recent years it has been observed that one of the important factors that influences the design and use of the register file is the overhead produced by the saving and restoring of registers in function calls and returns. For instance, Lunde [LUND77] affirms that the BLISS compiler spends 25% of its execution time to compile a program (Treesort) in call administration. Patterson and Séquin [PATT82] have weighted the relative frequency of high-level language statements to conclude that 12% of the statements are calls and that they correspond to 33% of the machine instructions and to 45% of the memory references (these numbers have been computed from the average number of instructions and references per statement generated by the C compilers for VAX-11, PDP-11, and MC68000). This is especially true for languages such as C which encourage the use of functions. Our measurements have shown that, on the aver-

age, one of each 25 executed data objects is a function call.

Some recent architectures have claimed that they have an efficient calling mechanism [STRE78, BERE82]: instead of having to specify instructions to explicitly perform register saving and restoring, this is done implicitly by the call and return instructions. Both the program size and the instruction memory traffic have been reduced because fewer instructions are needed to code high-level-function calls and returns, and fewer instructions have to be fetched. However, the data memory traffic is the same because the registers still must be saved and restored by the call and return instructions in the same way that they were saved and restored by simpler instructions. In VAX-11 the data memory traffic is even worse than PDP-11 because PDP-11 uses the frame pointer as argument pointer and VAX-11 have two registers for this purpose so that one more environment register has to be saved and restored per function call.

To avoid the register saving and restoring overhead some architects [JOHN82, DITZ82] have proposed to use a stack architecture. In a stack architecture variables and temporary results are stored in the stack so that no registers must be saved and restored during function calls. In this case, local simple variables have to be stored in the stack and we have already said in Section 1.4 that registers are more efficient than a cache to store local simple variables. Moreover, in Section 2.4 we show that register saving and restoring overhead is balanced by the reduction in the data memory traffic caused by the data objects when local simple variables are allocated to registers.

With the recent technological advancements it is possible to have a large number of registers on a single chip [SITE79] so that multiple register banks can be available in the processor. In this case, a new bank is made active on each function call and no registers have to be saved unless no more banks are available [SITE79, DANN79, LAMP82]. In this thesis we use the name *window* to refer to the part of the register set that is automatically allocated when a new function is entered. Our measurements confirm that a multiple-window register file is very effective to reduce the overhead caused by register saving and restoring (see Section 3.2). However, the resulting register files have a large number of registers, use a large chip area in VLSI implementations, increase the amount of processor context to be saved on context switching, and increase the processor cycle-time due to the long data busses [HENN84].

Most designs use fixed-size windows because of the simplicity in the implementation: C/70 [KRAL80, BBN81], RISC [PATT82, SEQU82], PYRAMID 90x [PTC83], and CELERITY C1200 [OLLE85]. Ditzel and McLellan [DITZ82] propose the use of variable-size windows to improve the use of registers, but this results in a complex and slower implementation (see Section 3.3). As a compromise that incorporates the advantages of both schemes, we propose an approach that uses a few (2 or 3) window sizes (Section 3.4). We use our measurements to

compare with the fixed-size window approach and to determine suitable window sizes (Section 3.7). We show that for equivalent performance the register file can be made significantly smaller (Section 3.8).

1.8 Thesis Organization

This thesis is organized as follows. Chapter 2 discusses the register usage for the purposes introduced in Section 1.5: local simple variables defined either by the programmer or by the compiler, registers to store the run-time process environment, registers to pass parameters, and registers for global simple variables. This chapter also considers the single window case to conclude that our measurements indicate that, in spite of the overhead in register saving and restoring, it is convenient to use registers for local simple variables.

Chapter 3 confirms the reduction in data memory traffic when multiple windows are available, compares fixed-size windows with the variable size case and non-overlapped windows with overlapped ones, presents the proposed approach of using a few different sizes, uses our measurements to select these sizes, discusses the size of the register file, and shows that in the proposed approach the size is significantly smaller than in the fixed-size window case. This chapter also considers the effects of exceptions and context switching on the register file and studies two different organizations: private register file and shared register file.

Chapter 4 discusses how the register file is organized using a circular buffer approach to implement windows of different sizes, compares this implementation with the existing organizations with fixed-size windows (C/70, RISC, PYRAMID 90x, and CELERITY C1200), and introduces a new organization for the register file that makes the bus size and the access time to the register file independent of its size, and equivalent to the bus size and the access time required for a single-window architecture.

CHAPTER 2
REGISTER USAGE

2.1 Introduction

This chapter discusses how registers are used to store local simple variables either defined by the programmer or by the compiler, environment state information, and global simple variables. Except when it is explicitly mentioned, local simple variables also include the parameters passed to the function. This chapter also presents how registers can be used to pass parameters.

Local simple variables defined by the programmer correspond not only to data of a fundamental data type (integer, character, unsigned, ...), but also to pointers to variables of any type, i.e., any variable that can be stored in a register. The C programmer uses two different storage classes to define local simple variables: automatic and register. The definition of register variables is used to help the compiler to perform register allocation.

The compiler can generate some local simple variables either to perform expression evaluation or to optimize program execution. Let us call *temporary variables* the local simple variables defined by the compiler to perform expression evaluation, and *optimizing variables* the local simple variables defined by the compiler to optimize program execution. These variables are either of a fundamental data type or pointers to variables of any type. The difference between them is that temporary variables exist only during expression evaluation (i.e., they are alive per statement basis) and optimizing variables exist across statements.

There are two advantages for using registers to allocate local simple variables either defined by the programmer or by the compiler. First, program size is reduced because register addresses are shorter than memory addresses. Second, there is less data memory traffic generated during program execution because registers have faster access than memory. Section 2.2 shows the usage of local simple variables done by the programmer and the number of local simple variable defined by the programmer as an upper bound of the number of registers required per function for this purpose.

On the other hand, there is a main drawback in allocating local simple variables to registers: the complexity of the compiler increases. The register allocation policy is a complex task to be performed by the compiler due to two factors: the alias problem and the register saving and restoring overhead. If the

compiler wants to move a local simple variable from the stack to registers, it has to be sure that there is no alias (i.e., the variable cannot be accessed through a pointer) because, otherwise, two copies of the same variables could be updated simultaneously. The alias problem and its solutions are discussed in Section 2.3.

The second factor is given by the register saving and restoring overhead. If the compiler moves local simple variables to registers knowing that they have no alias, then every time that a function is called the registers must be saved, and when a function returns, they must be restored. This increases the data memory traffic so that compiler writers want to make sure that even if the local simple variable has no alias, its usage in the function (deduced from the static number of references to the variable) compensates for the traffic overhead generated. Section 2.4 considers the data memory traffic generated in the single window case to conclude that our measurements indicate that, in spite of the overhead in register saving and restoring, it is convenient to use registers for local simple variables. Moreover, McKusick [MCKU84] affirms that "by eliminating entry and exit costs (to function calls with multiple windows), the register allocator can more readily use the registers without needing to worry about whether the use will have enough payoff to cover the entry and exit cost." Multiple windows are discussed in the next chapter.

From the previous discussion we can conclude that the effectiveness of register usage depends on the characteristics of the programs to execute, the characteristics of the compiler which translates the high-level language program, and the characteristics of the processor. Each one can be characterized as follows:

(1) The programs can be characterized by three factors: the usage of local simple variables defined by the programmer with respect to the total number of data object executed by the program, the number of local simple variables defined for each function, and the frequency of performing function calls. Section 2.2 discusses the first two factors. The frequency of function calls has already been given in Section 1.7: on the average, one of every 25 executed data objects is a function call.

(2) The compiler is characterized by the register allocation policy that is used to decide which variables are allocated to registers and by the number of temporary and optimizing variables that the compiler generates. Section 2.5 discusses three different allocation policies, Section 2.6 presents the number of registers required to perform expression evaluation, and Section 2.7 considers the number of registers required for optimizing variables.

(3) The processor is characterized by the number of registers available and the cost of register saving and restoring given by the number of registers to save. Section 2.4 shows the register saving and restoring overhead for three different register allocation policies in the single window case: (i) all local simple variables are

allocated to processor registers, (ii) only explicitly defined register variables are, and (iii) no local simple variables are. The number of registers in the processor is only considered in the mentioned section as the number of registers to be saved when a function is called. Chapter 3 considers again this cost for the multiple window case and deduces the number of registers that should be available in the processor and how they should be organized.

Finally, this chapter also presents how many registers are required to pass parameters (Section 2.8) and to store the run-time process environment state information (Section 2.9), and how memory mapped registers can be used to store some global simple variables (Section 2.10).

2.2 Local Simple Variables Used by the Programmer

In this section we discuss first the usage of local simple variables defined by the programmer with respect to the total number of data objects that the program generates and executes. We confirm the results already given by other authors [PATT82, COOK82] that local simple variables are the most common variables used by the programmer. Second, we also comment on the use of variables of the register class done by the programmer. We show that applications programmers use fewer register variables definitions that system programmers so that we cannot expect to allocate registers only based on the storage class defined by the programmer to obtain a good generated code as some register allocation policies are doing (see Section 2.5). Finally, we consider the number of local simple variables defined by the programmer and confirm again that the most frequent called functions are the ones with a small number of defined local simple variables.

2.2.1 Local Simple Variable Usage

In this subsection we discuss the usage of local simple variables defined by the programmer with respect to the total number of data objects that the program generates and executes. This usage is given for the four measured programs mentioned in Section 1.3.: CIFPLOT (only static measurements), NROFF, SORT, and VPCC; the column AP (ALL PROGRAMS) indicates their totals. Table 2.1 shows the static and dynamic usage of automatic and register variables for each programs, i.e., for the data objects that can be allocated in registers. Data objects are divided into two categories depending on the storage class: automatic or register.

The first row shows the percentage of data objects which are local simple variables used as operands (local simple variables with and without autoincrement or autodecrement operators), the second shows the percentage of data objects that are local simple variables used to compute the address of an operand (pointers to simple variables with and without autoincrement or autodecrement operators and pointers to a field of a structure), and the third the addition of

both rows. Parameters are accounted as part of the local simple variables of the function.

local sv used as	storage class	static (%) CIFPLOT	NROFF	SORT	VPCC	AP	dynamic (%) NROFF	SORT	VPCC	AP
operand	auto	22.7	8.5	19.2	9.2	14.3	7.5	35.9	8.9	9.4
	register	5.1	20.0	15.5	14.3	12.4	32.4	16.8	29.2	30.2
address comp.	auto	8.4	0.6	5.6	3.0	4.3	0.2	9.4	3.2	2.2
	register	2.3	1.8	8.5	14.1	7.3	0.6	8.6	13.9	8.2
TOTAL	auto	31.1	9.1	24.8	12.2	18.7	7.7	45.3	12.1	11.5
	register	7.4	21.8	23.9	28.4	19.7	33.0	25.4	43.1	38.3

Table 2.1. Local Simple Variable Usage

From the table we see that 40% of the executed data objects operate directly a local simple variable (with or without increment or decrement operators), and 10% of the executed data objects require a local simple variable to compute the address of the operand. From this, we can conclude that if the compiler allocates all the local simple variables in registers and there are enough registers to store them, then

a) The static measurements show that 38% of the data objects are referenced with a register address.

b) The dynamic measurements show that 50% of the data objects are found in registers during program execution.

However, if the compiler only allocates register variables explicitly defined by the programmer, then

a) Only half of the local simple variables (19% of the data objects) are referenced with a register address.

b) Only 38% of the data objects are found in registers during program execution.

Furthermore, our measurements show that 30% of the data objects does not generate any data memory traffic because they are either integer constants (26.7%) or function calls (3.8%). Therefore, storing as many local simple variables as possible in registers should reduce significantly the size of the programs and the data memory traffic. Section 2.4 discusses in more detail data memory traffic generated when local simple variables are completely allocated, parcially allocated, and no allocated to registers.

2.2.2 Automatic versus Register Variable Definitions

In this subsection we comment on the use of register variables done by the programmer. If the programmer utilizes register definitions in its simple variable declarations, then register allocation will be greatly simplified because only variables defined by the programmer are allocated to registers. However, for this to be effective the programmer should know how many registers are available in the processor, the allocation policy that the compiler uses, and the best candidates to be stored in registers. This results in a program that is not machine independent and that runs efficiently only in the machine that has been generated for. Moreover, as can be seen from Table 2.2 register variables are not used by application programmers so that efficiency in register allocation depends on the compiler. Therefore, the compiler should be responsible for optimizing the generated code.

	CIFPLOT	NROFF	SORT	VPCC	ALL PROGRAMS
Declared functions	274	226	21	252	773
Total no. of registers	89	260	30	412	791
Regs. per function	0.32	1.15	1.43	1.63	1.02
Total no. of automatic	794	198	61	719	1772
Auto. per function	2.95	0.88	2.90	2.85	2.31
Ratio reg.:auto.	0.11	1.31	0.49	0.57	0.44

Table 2.2. Local Simple Variables Defined by the Programmer

Table 2.2 gives the number of local simple variable that the programmer is defining as register and as automatic for the four measured programs (see Section 1.3). It can be easily deduced that

(1) System programmers use the register allocation definition to optimize program execution. The best usage is for compiler writers: 1.63 registers per function; the number is not higher maybe because several functions were written originally for PDP-11 and the Portable C Compiler for PDP-11 only has three registers available for register defined variables. The ratio of registers versus automatic declarations is 0.72 for the three system programs (NROFF, SORT, and VPCC).

(2) Application programmers use fewer register definitions. CIFPLOT has on the average 0.32 registers per function and the ratio of register versus automatic declarations is 0.11.

How general are these results? This is difficult to say. My experience says that programmers who have started as assembly language programmers use more registers than programmers who have started with another high level language (PASCAL, for instance). But this is impossible to quantify. Let us see how the programmer uses register definitions in what is consider to be representative C

programs: the benchmarks. Eleven benchmark programs have been measured: five (E, F, H, I, K) were developed at Carnegie-Mellon University to evaluate different computer architectures [GRAP81], and they were coded in C by Piepho [PIEP81]; the ackermann's function (ack) [WICH76]; two version of the puzzle benchmark, one using pointers (pptr) and the other using subindexes (psub) [BEEL84]; the quicksort benchmark (sort); the Sieve of Eratothenes benchmark (siev) [GILB81]; and the towers of Hanoi benchmark (tow). These benchmarks were provided by David Patterson and they have been used to compare RISC with other architectures [PIEP81, PATT82a].

	E	F	H	I	K	ack	pptr	psub	sort	siev	tow	tot
no. ftn.	2	2	2	2	2	2	5	5	6	1	2	31
no. reg.		5			7		24		16		1	53
average		2.5			3.5		4.8		2.7		0.5	1.7
no. auto.	16	9	7	16	13	2		11	15	5	4	98
average	8.0	4.5	3.5	8.0	6.5	1.0		2.2	2.5	5.0	2.0	3.2
ratio		0.6			0.5				1.1		0.3	0.5

Table 2.3. Register Usage by Benchmark Programmer

Table 2.3 gives the register usage by the programmer. Six benchmark programs out of 11 do not have register definitions at all. On the average the ratio of register definitions to automatic definitions is 0.5. This ratio is similar to the system programs VPCC and SORT. This high ratio is caused basically by the pointer version of the puzzle (pptr) benchmark (all local simple variables have been defined register) and the sort benchmark (the most used local simple variables are defined register). If these two benchmarks were not considered because of their high number of register declarations, then the ratio would be 0.2, closer to the one given by the application program CIFPLOT. Therefore, in general, application programmers use few register definitions.

2.2.3 Number of Local Simple Variables per Function

Now we discuss the number of local simple variables that the programmer defines in each function. The number of registers required to store these local simple variables will depend on the register allocation policy. Three different allocation policies are presented in Section 2.5. However, we can use the number of local simple variables defined as an upper bound for the number of registers required per function for this purpose.

Table 2.4 gives the percentage of functions with up to 3, 6, 8, and 14 local simple variables defined in the function, including parameters. The percentages of functions are given statically and dynamically for the four measured programs. These numbers correspond to the actual number of registers used to store local simple variables defined by the programmer by the Portable C Compiler for

PDP-11 (3), for VAX-11 (6), and for RISC (14)*, and by the RIDGE C Compiler for RIDGE 32 (8).

Programs	Number of registers available for local variables							
	static				dynamic			
	3	6	8	14	3	6	8	14
CIFPLOT	70.1	87.5	93.6	99	-	-	-	-
NROFF	83.6	93.8	98.7	100	89.2	98.6	99.9	100.0
SORT	61.9	81	85.7	95.2	77.9	79.5	79.5	80.6
VPCC	69.5	88.9	96.0	98.8	59.0	72.8	92.7	97.8
ALL PROG	73.6	89.7	95.7	99.1	75.4	86.6	96.2	98.6

Table 2.4. Percentage of Defined Local Simple Variables

From the table we can deduce that 99% of functions have no more than 14 local simple variables. However, we can also observe that the most frequent called functions are the ones with a small number of local simple variables: 75% of the called functions have up to three local simple variables. Also that the number of local simple variables is not equally distributed among the functions: 75% have up to three, 12% have between four and six, 9% have seven or eight, and 3% have between nine and fourteen.

In conclusion, if local simple variables are stored in registers, data memory traffic is reduced because of their frequent usage. Also code size is reduced because shorter addresses are required to refer to a variable stored in a register than to a variable stored in the stack. Moreover, the compiler should allocate simple variables to registers independent of their storage class (register or automatic) as defined by the programmer. That is, we cannot expect to allocate registers only based on the storage class defined by the programmer to obtain a good generated code. Finally, we have shown that the most frequent called functions are the ones with a small number of local simple variables (75% of the functions have up to three local simple variables) and that the number of local simple variables defined by the programmer is not very large (99% of the function have up to fourteen local simple variables).

*In reality the Portable C Compiler only uses nine registers for local simple variables. However, this could be expanded to the fourteen registers allowed by the architecture (see Section 3.7.3).

2.3 Alias Problem

One of the problems encountered by compilers in the allocation of local simple variables to registers is the alias problem. A name is an alias of a second one when both of them refer to the same memory location. The language definition in C does not allow the definition of two names that refer to the same memory location, i.e., there is no construct similar to EQUIVALENCE of FORTRAN. However, two different names can refer to the same memory location because

(a) One is the definition name of this location and the other is a pointer to it. For example, for the variables and functions in Figure 2.1 the integer pointer *p* of the function *g* and the global integer *i* refers to the same memory location. The execution of the program (independent of its purpose) would not be correct if the compiler moves the global integer variable *i* to a register before entering the **for** loop.

```
int i, a[N];
f ()
{
    g(&i);
}
g (p)
int *p;
{
        int j;
        for (j = 0; j < N; j++)
                if (a[j] > i)
                        *p += a[j];
}
```

Figure 2.1. Example of Alias Problem

(b) Two pointers are pointing to the same memory location. In this case there are two local or global pointers that contain the same address, i.e., they are referring to the same memory location.

This phenomenon is explicitly caused by the programmer with or without intent, but the compiler must generate correct code for it. This phenomenon is not only typical for the C language, but also for block-structured languages as ALGOL, PASCAL, and ADA. If the compiler decides to move temporarily a copy of the value of a variable that is in the stack to a register and this variable has an alias, then this copy is valid until either the next function call or the next expression evaluation which includes a pointer indirection. This is because the pointer indirection or the called function can update the value of the variable that is in the stack so that the copy that is in the register is not any longer valid. Also the pointer indirection or the called function can read the value from the stack after

its copy has been updated in the registers.

If the language had few function calls and few pointer indirections, then the lifetime of a variable in a register would be large enough to compensate the work done by the compiler to keep track of which variables are in registers. However, function calls in C and, in general, in modern programming languages are very frequent so that the lifetime of a variable in a register is short. Our measurements show that one of each 25 executed data objects is a function call and that one of each 50 executed data objects is a pointer indirection. The period of life of a register has been measured by Lunde [LUND77] for six algorithms written in ALGOL, BASIC, BLISS, FORTRAN, and MACRO and executed in DEC-10; it is 11.9 instructions ranging from 4 to 24 instructions. Therefore, variables are not moved temporarily to a register because their lifetime is short.

However, if the variable has no alias, then the copy that is in the register is still valid after a pointer indirection or a function call. There are four solutions to the alias problem:

(1) The easiest one is not to use registers at all. This is the solution implemented by stack architectures. It results in high memory traffic and slow access to the variables.

(2) The compiler only allocates to registers variables explicitly defined of register storage class by the programmer. Also the compiler does not move temporarily any automatic variable to a register so that only one copy of a variable exists at any point during program execution. The alias problem is solved because the address operator & cannot be applied to a register variable, i.e., they cannot be accessed through a pointer, and only a copy for each variable exists at any time either in the stack if it is an automatic variable or in a register if it is a register variable. The disadvantage is that register usage is far from being optimal as has been discussed in the previous section. This is the policy used, for instance, by the Portable C Compiler [JOHN79] for PDP-11 and VAX-11.

(3) The compiler allocates a local automatic variable to a register if the programmer does not apply the address operator to it. In this case, there is no other name that can refer to it by indirection so that if the compiler moves temporarily a copy of the variable to a register, then we can guarantee that the variable has no alias. This solution is implemented, for instance, in the VMS C Compiler [ANKL82]. The number of local simple variables that cannot be allocated to registers because the address operator is applied to them is insignificant: our measurements show that only 0.2% of the data objects are local simple variables

with the address operator and they account for 0.0013%! of the executed objects*. Consequently, the alias problem can be solved by an optimizing compiler and almost all the local simple variables can be allocated to registers if there are enough registers available.

(4) The registers are mapped to memory so that both a register address and a memory address refer to the same location. In this case, register and automatic variables can be allocated to registers when they are defined (assuming that there are enough registers to store all local variables). If the compiler finds out later that the address operator is applied to a local simple variable which has been allocated to a register, then the mapped address is computed so that both the register number and the address refer to the same memory location. The memory unit detects when the processor is issuing a memory address to a location that refers to a register and the appropriate access is generated (Section 4.2.5 shows how this is implemented). Memory mapped registers are also used for other block-structured languages as a mechanism to access to a variable defined in a higher lexicographical level. However, Tanenbaum has already detected that this is done very infrequently [TANE78].

Memory mapped registers require a new set of memory addresses for each activation record. The reason is that when a function is called and its new activation record is built, the contents of the actual registers are saved and, thus, their mapped address are no longer valid. Only part of the activation record needs a new set of memory mapped addresses, the part that resides in registers and that must be saved. Consequently, the activation record is divided in two parts: the part that is stored in registers (the local variables defined by the programmer allocated to registers and the variables required by the compiler) and the part that is stored in the stack frame (local variables defined either by the programmer or by the compiler that cannot be allocated to registers and the local structures and arrays). Thus, two stacks (with their associated stack pointers) are required to implement memory mapped registers as is shown in Figure 2.2. Memory mapped registers have been proposed by Patterson and Séquin [PATT82], although neither RISC I [PEEK83] nor RISC II [KATE83] have implemented them. PYRAMID 90x [PTC83] has implemented memory mapped registers, although the cost of the implementation cannot be justified by the gain detected in our measurements.

In conclusion, both (3) and (4) provide solutions to the alias problem. (3) solves the alias problem by having an optimizing compiler select which variables can be allocated to registers and does not require any architectural support. Our measurements show that the number of local simple variables that cannot be allocated to registers because the address operator is applied to them is insignificant

*Since these numbers are very low, no attempt was made for computing exactly how many variables cannot be allocated to registers because the address operator is applied to them.

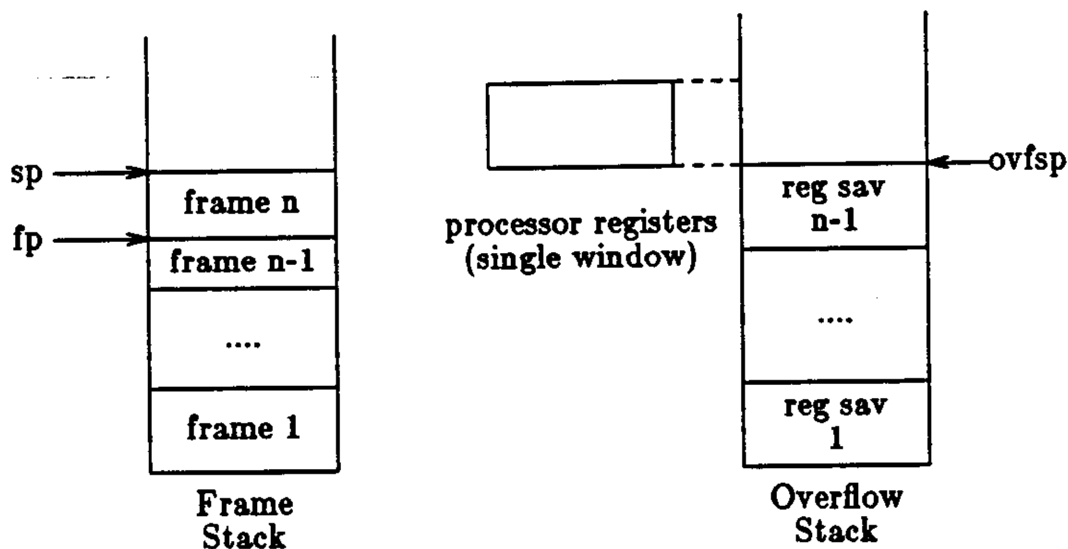


Figure 2.2. Activation Record Partition with Memory Mapped Registers

because only 0.2% of the data objects are local simple variables with the address operator and they account for 0.0013%! of the executed objects. In (4) the alias problem does not exist because the compiler can always generate a memory address to refer to an operand stored in a register; in this case, a non-optimizing compiler can allocate all local simple variables to registers getting a good optimized code and without having to worry about the alias problem. Although both of them can be equally efficient, we must balance the problems that the compiler has to allocate local simple variables to registers with and without architectural support and the influence of the hardware support over the processor cycle time. Section 2.5 reviews how compilers can solve the alias problem without architectural support and Section 4.2.5 discusses the requirements to implement memory mapped registers with a circular-buffer register file. A shift-register file does not support memory mapped registers (see Section 4.3).

2.4 Data Memory Traffic Overhead

The measurements presented in this Section show that for the single-window case registers should be used to store local simple variables to reduce memory traffic, in spite of the memory traffic overhead generated by register saving and restoring on function calls.

The single-window case is studied because it seems that some compilers do not use registers to store local simple variables due to the register saving and restoring overhead. This is shown, for instance, in a study made by Emer and Clark [EMER84] where they have monitored the performance of a VAX-11/780 under a specific timesharing workload. The number of reads and writes caused by call

and return instructions is 4.14 and 4.03 respectively, i.e., on the average no registers are used across function calls, only three environment registers (the program counter, the frame pointer, the argument pointer) and a mask (with some information that is not relevant to our discussion) are saved and restored [DEC79].

To compute the total data memory traffic we have to add to the memory traffic generated by the data objects, the one generated by parameter passing and by register saving and restoring. Since our measurements reflect the dynamic characteristics of the C programs independently of the machine on which they are executed, the data memory traffic is "computed" instead of "measured." The data objects have been measured dynamically for the three programs mentioned in Section 1.3. The number of data memory accesses have been computed according to the storage class associated to each data object type.

Subsection 2.4.1 shows the data memory traffic caused by executed data objects and discusses how this have been computed. Subsection 2.4.2 shows the total memory traffic once the overhead caused by parameter passing and register saving and restoring is included.

2.4.1 Data Memory Traffic Caused by Data Objects

This section shows the data memory traffic caused by the data objects for the measured programs: NROFF (Table 2.5), SORT (Table 2.6), VPCC (Table 2.7), and their totals (ALL PROGRAMS; Table 2.8). Since we want to determine the data memory traffic for different allocation policies and for different number of registers used, we consider the following three cases:

(a) Only explicitly defined register variables are allocated to processor registers (and there are enough processor registers for all register variables). This type of register allocation is performed by the Portable C Compiler [JOHN79] for VAX-11. This case is called "case (a)."

(b) All local simple variables defined by the programmer are allocated to registers. This case corresponds to a register allocation policy in which all local simple variables are allocated to registers without register sharing, i.e., even if two local simple variables have disjoint lifetimes they are allocated to different registers*. This type of register allocation is performed by the Portable C Compiler for RISC [MIRO82]. This case is called "case (b)."

*If two or more variables share the same register, then less registers are required than local simple variables defined in the function. Since we did not know in this case how many registers should be saved, on the average, when a function is called, then we would not be able to compute the register saving and restoring overhead in Subsection 2.4.2. For this reason we must consider that each variable is allocated to a different register, even if they have disjoint lifetimes. However, a sharing allocation policy would also correspond to this case when enough registers are available.

NROFF		C Data Objects No.	%	case (a) No.	%	case (b) No.	%	case (c) No.	%
SV	global	13813498	19.1	1.0	45.0	1.0	56.6	1.0	23.8
	auto	5015869	6.9	1.0	17.1	-	-	1.0	9.1
	register	21489449	29.7	-	-	-	-	1.0	38.8
ARY	global	1755907	2.4	1.0	6.0	1.0	7.5	1.0	3.2
	auto	5441	0.0	1.0	0.0	1.0	0.0	1.0	0.0
	register	721	0.0	1.0	0.0	1.0	0.0	2.0	0.0
PTR	global	91927	0.1	2.0	0.6	2.0	0.8	2.0	0.3
	auto	113303	0.2	1.8	0.7	0.8	0.4	1.8	0.4
	register	225200	0.3	1.0	0.8	1.0	1.0	2.0	0.8
→	global	31839	0.0	2.0	0.2	2.0	0.3	2.0	0.1
	auto	-	-	-	-	-	-	-	-
	register	33781	0.0	1.0	0.1	1.0	0.1	2.0	0.1
v++	global	2137062	3.0	2.0	14.6	2.0	18.4	2.0	7.7
	auto	411449	0.6	2.0	2.8	-	-	2.0	1.5
	register	1972812	2.7	-	-	-	-	2.0	7.1
*++	global	769162	1.1	3.0	7.9	3.0	9.9	3.0	4.2
	auto	18535	0.0	3.0	0.2	1.0	0.1	3.0	0.1
	register	218350	0.3	1.0	0.7	1.0	0.9	3.0	1.2
as[].f	global	918522	1.3	1.0	3.1	1.0	3.9	1.0	1.7
	auto	-	-	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
&	global	196262	0.3	-	-	-	-	-	-
	auto	3	0.0	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
FTN		3277421	4.5	-	-	-	-	-	-
Integer constants		19814108	27.4	-	-	-	-	-	-
Strings		18516	0.0	-	-	-	-	-	-
TOTALS		72329137		29248565		23259541		55380040	

Table 2.5. Data Memory Traffic per Data Object (NROFF)

(c) No local simple variables are allocated to processor registers, i.e., all local simple variables are allocated to the stack. This case would correspond to the register allocation policy performed by the Portable C Compiler for VAX-11 if the programmer did not use any register variable definition at all. This case is called "case (c)."

The data memory traffic has been computed considering the storage class (global, automatic or register) of the data object and the number of memory accesses that the data object generates. The values for the three cases are given relative to the number of data objects. The following considerations have to be taken into account to compute the data memory traffic from the measured data objects:

SORT		C Data Objects No.	%	case (a) No.	%	case (b) No.	%	case (c) No.	%
SV	global	374963	5.4	0.3	1.9	0.3	6.0	0.3	1.3
	auto	2160737	31.3	1.0	40.4	-	-	1.0	29.0
	register	868766	12.6	-	-	-	-	1.0	11.7
ARY	global	4525	0.1	1.0	0.3	1.0	0.1	1.0	0.1
	auto	-	-	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
PTR	global	27724	0.4	1.0	0.5	1.0	1.7	1.0	0.4
	auto	316795	4.6	2.0	11.8	1.0	19.0	2.0	8.5
	register	531161	7.7	1.0	9.9	1.0	31.9	2.0	14.3
→	global	-	-	-	-	-	-	-	-
	auto	59912	0.9	2.0	2.2	1.0	3.6	2.0	1.6
	register	6732	0.1	1.0	0.1	1.0	0.4	2.0	0.2
v++	global	79405	1.2	3.0	4.5	3.0	14.3	3.0	3.2
	auto	314587	4.6	2.1	12.3	0.2	3.2	2.1	8.8
	register	289949	4.2	0.0	0.0	0.0	0.0	2.0	7.8
*++	global	-	-	-	-	-	-	-	-
	auto	271118	3.9	3.0	15.2	1.0	16.3	3.0	10.9
	register	57611	0.8	1.0	1.1	1.0	3.5	3.0	2.3
as[].f	global	-	-	-	-	-	-	-	-
	auto	-	-	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
&	global	29957	0.4	-	-	-	-	-	-
	auto	-	-	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
FTN		114701	1.7	-	-	-	-	-	-
Integer constants		1389980	20.1	-	-	-	-	-	-
Strings		6	0.0	-	-	-	-	-	-
TOTALS		6898629		5349663		1667271		7451441	

Table 2.6. Data Memory Traffic per Data Object (SORT)

(1) Simple variable (SV) objects do not generate any memory access if the object is stored in a register, and one memory access, otherwise. However, there are some data objects that are considered as simple variables, but that instead of fetching their contents, their address has to be computed. This occurs, for instance, when a name for a structure or an array appears alone, i.e., without any field or element modifiers. This results that the traffic for some global variables is less than the number of objects because to compute the address of a data object no data memory traffic is involved.

(2) Array objects (ARY) generate one memory access, except when the address of an array has been passed as a parameter. In this case, if the address is in the stack (i.e., is stored in an automatic variable for the case (a)), then it should be fetched first to compute the address of the element to access. For this reason,

the traffic is greater than number of objects.

(3) Pointers to simple variables (PTR) or to a field of a structure (→) require two accesses if the pointer is in memory or only one if the pointer is in a register. The exceptions in this case are pointers to structures (used to compute the structure address) and the pointers to functions (used to store the address of the function and, therefore, only the address is loaded, not the datum where the address is pointing to).

VPCC		C Data Objects No.	%	case (a) No.	%	case (b) No.	%	case (c) No.	%
SV	global	3404162	4.3	0.8	8.3	0.8	11.9	0.8	3.8
	auto	6842709	8.6	1.0	20.9	0.0	0.0	1.0	9.5
	register	19502500	24.5	-	-	-	-	1.0	27.2
ARY	global	3704221	4.7	1.0	11.3	1.0	16.2	1.0	5.2
	auto	826	0.0	1.0	0.0	1.0	0.0	1.0	0.0
	register	137424	0.2	0.2	0.1	0.2	0.1	1.2	0.2
PTR	global	39743	0.0	2.0	0.2	2.0	0.3	2.0	0.1
	auto	463077	0.6	1.1	1.5	0.1	0.1	1.1	0.7
	register	1914680	2.4	0.9	5.3	0.9	7.5	1.9	5.1
→	global	141505	0.2	2.0	0.9	2.0	1.2	2.0	0.4
	auto	2100681	2.6	2.0	12.8	1.0	9.2	2.0	5.9
	register	9088362	11.4	1.0	27.8	1.0	39.8	2.0	25.4
v++	global	103006	0.1	2.1	0.6	2.1	0.9	2.1	0.3
	auto	226840	0.3	2.0	1.4	0.0	0.0	2.0	0.6
	register	3721156	4.7	0.5	5.5	0.5	7.9	2.2	11.7
*++	global	5896	0.0	3.0	0.1	3.0	0.1	3.0	0.0
	auto	22452	0.0	3.0	0.2	1.0	0.1	3.0	0.1
	register	814422	1.0	1.0	2.5	1.0	3.6	3.0	3.4
as[].f	global	76046	0.1	1.0	0.2	1.0	0.3	1.0	0.1
	auto	-	-	-	-	-	-	-	-
	register	113228	0.1	1.0	0.3	1.0	0.5	2.0	0.3
&	global	2892227	3.6	-	-	-	-	-	-
	auto	196213	0.2	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
FTN		2719830	3.4	-	-	-	-	-	-
Integer constants		21219000	26.7	-	-	-	-	-	-
Strings		100406	0.1	-	-	-	-	-	-
TOTALS		79550612		32705235		22816695		71627850	

Table 2.7. Data Memory Traffic per Data Object (VPCC)

(4) Simple variables that are incremented or decremented (v++) require also two accesses if the object is in memory, and none if the object is in a register. The exception in this case is the objects of type *(p→f)++* that require three accesses if the pointer is in memory, or two if it is in a register.

(5) Pointers to simple variables that are auto-incremented or auto-decremented (*++) require three accesses if they are in memory, or one if they are in registers.

(6) Fields of an element of an array of structures (as[].f) require only one access, except in the case in which its address has been passed as a parameter.

(7) Addresses of objects (&) are solved either computing the address or loading a constant with their memory address. In any case, no data memory traffic is involved.

(8) Integer constants are loaded directly from the instruction register; therefore, no data memory traffic is involved.

(9) For strings, their address is part of the instruction and, thus, no data memory traffic is involved.

Other data objects have not been included in this table because no occurrence has been detected (local structures, pointers to a field of a structure with auto-increment or auto-decremented operators, and float constants).

It can be seen that in case (b) traffic caused by automatic variables has been either eliminated (case 1) or reduced (cases 2 to 6) because all local simple variables are allocated to registers. Note that in this case global simple variables (with or without auto-increment or auto-decrement operators) account, on the average, for 45% of the data memory traffic; Section 2.10 shows how this traffic can also be reduced.

Table 2.9 summarizes the traffic caused by data objects for the three cases. The more simple variables the program has, the more traffic is introduced because these variables must be brought from memory every time they are referenced. For example, SORT using 49% of simple variable objects (31% are automatic variables) introduces an overhead of 221% when only register variables are allocated to registers with respect to when all local simple variables are. If no local simple variables are allocated, then the overhead is in this case of 347%!

In conclusion, if only register variables are allocated to registers (case (a)), then the data memory traffic generated by the data objects is 41% more with respect to the case in which all local simple variables are allocated to registers (case (b)). If no local simple variables are allocated to registers (case (c)), then the data traffic is 100% more with respect to the case in which only the register variables are allocated to registers and is 182% more with respect to the case in which all local simple variables are allocated to registers. These numbers do not include the data memory traffic generated by parameter passing and register saving and restoring. This is discussed next.

ALL PROGRAMS		C Data Objects No.	%	RISC I No.	%	VAX-11 No.	%	VAX-11 No.	%
SV	global	17592623	11.1	0.9	23.8	0.9	33.5	0.9	11.9
	auto	14019315	8.8	1.0	20.8	0.0	0.0	1.0	10.4
	register	41860715	26.4	-	-	-	-	1.0	31.1
ARY	global	5464653	3.4	1.0	8.1	1.0	11.4	1.0	4.1
	auto	6267	0.0	1.0	0.0	1.0	0.0	1.0	0.0
	register	138145	0.1	0.2	0.0	0.2	0.0	1.2	0.1
PTR	global	159394	0.1	1.8	0.4	1.8	0.6	1.8	0.2
	auto	893175	0.6	1.5	2.0	0.5	0.9	1.5	1.0
	register	2671041	1.7	0.9	3.7	0.9	5.2	1.9	3.8
→	global	173344	0.1	2.0	0.5	2.0	0.7	2.0	0.3
	auto	2160593	1.4	2.0	6.4	1.0	4.5	2.0	3.2
	register	9128875	5.7	1.0	13.6	1.0	19.1	2.0	13.6
v++	global	2319473	1.5	2.0	7.0	2.0	9.9	2.0	3.5
	auto	952876	0.6	2.0	2.9	0.1	0.1	2.0	1.4
	register	5983917	3.8	0.3	2.7	0.3	3.8	2.2	9.6
*++	global	775058	0.5	3.0	3.5	3.0	4.9	3.0	1.7
	auto	312105	0.2	3.0	1.4	1.0	0.7	3.0	0.7
	register	1090383	0.7	1.0	1.6	1.0	2.3	3.0	2.4
as[].f	global	994568	0.6	1.0	1.5	1.0	2.1	1.0	0.7
	auto	-	-	-	-	-	-	-	-
	register	113228	0.1	1.0	0.2	1.0	0.2	2.0	0.2
&	global	3118446	2.0	-	-	-	-	-	-
	auto	196216	0.1	-	-	-	-	-	-
	register	-	-	-	-	-	-	-	-
FTN		6111952	3.8	-	-	-	-	-	-
Integer constants		42423088	26.7	-	-	-	-	-	-
Strings		118928	0.1	-	-	-	-	-	-
TOTALS		158778378		67303463		47743507		134459331	

Table 2.8. Data Memory Traffic per Data Object (ALL PROGRAMS)

% overhead	NROFF	SORT	VPCC	ALL PROGRAMS
case (a) vs. case (b)	25.7	220.9	43.3	41.0
case (c) vs. case (a)	89.3	39.3	119.0	99.8
case (c) vs. case (b)	138.1	346.9	213.9	181.6

Table 2.9. Traffic Caused by Data Objects

2.4.2 Overhead for Single Windows

For a single-window architecture style, we want to determine the register saving and restoring cost for different allocation policies and for different number of registers used. Therefore, we consider the three cases mentioned in the previous Subsection: (1) only explicitly defined register variables are allocated to processor registers (this case is called "SW (a)"); (2) all local simple variables defined by the programmer are allocated to registers (this case is called "SW (b)"); and (3) no local simple variables are allocated to processor registers (this case is called "SW (c)").

For the computation of the overhead caused by register saving and restoring we consider that at least four elements have to be pushed onto the stack when the function is called; these elements correspond to the three environment registers and the mask mentioned (at the beginning of Section 2.4) for VAX-11. In addition, the registers being used to store local simple variables have also to be saved. For the computation of the overhead caused by parameter passing we consider the usual approach of passing them through the stack (see Section 2.8) so that one memory reference is made for each parameter.

The data memory traffic is given in Table 2.10. From the table we see that, on the average, 1.41 register variables per function are defined by the programmer. Consequently, including the saving and restoring of the three environment registers and the mask, results in 10.82 references to memory being generated per function call. Therefore, SW (a) has, on the average, 114% more data memory traffic than that strictly caused by the programmer when parameter passing and register saving and restoring is considered.

When all local simple variables are allocated to registers (i.e., for SW (b)), the data memory traffic generated by the data objects is reduced by 29% with respect to the one generated for SW (a), but the register saving and restoring traffic is 36% more because, on the average, 3.37 local simple variables are allocated to registers (this number results from the 1.41 register variables and the 1.96 automatic variables defined on the average per function; see Table 2.10). Thus, the overhead generated by register saving and restoring and parameter passing is 214% instead of the previous 114%. Consequently, the reduction in data object traffic is balanced by the increase in register saving and restoring overhead, resulting in a total memory traffic that is very similar for both cases.

On the other hand, if no local simple variables are allocated to registers (i.e., for SW (c)), then the data memory traffic generated by the data objects is twice the one generated for the case SW (a), but the register saving and restoring traffic is 26% less because only eight memory references are generated for each function call (the saving and restoring of the three environment registers and the mask). This results in 34% more data memory traffic than for case SW (a).

SINGLE WINDOW	NROFF	SORT	VPCC	ALL PROGRAMS
No. function calls	3309538	142398	3041209	6493145
register vars. per ftn.	1.15	1.43	1.63	1.41
Automatic vars. per ftn.	0.88	2.90	2.85	1.96
case (a): only reg. vars.				
data memory traffic	64959158	7067770	71554324	143958384
data objects	29248565	5349663	32705235	67303463
parameters	1622352	171665	4605076	6399093
registers	34088241	1546442	34244013	70255828
% overhead (tot./d.obj.)	122	32	119	114
case (b): all local vars.				
data memory traffic	64794921	4211286	79000676	149851557
data objects	23259541	1667271	22816695	47743507
parameters	1622352	171665	4605076	6399093
registers	39913028	2372350	51578095	95708957
% overhead (tot./d.obj.)	179	153	246	214
SW(b) vs. SW(a) overhead	0	-40	10	4
case (c): no local vars.				
data memory traffic	83478696	8762290	100562598	192803584
data objects	55380040	7451441	71627850	134459331
parameters	1622352	171665	4605076	6399093
registers	26476304	1139184	24329672	51945160
% overhead (tot./d.obj.)	51	18	40	43
SW(c) vs. SW(a) overhead	29	24	41	34
SW(c) vs. SW(b) overhead	29	108	27	29

Table 2.10. Overhead for Single Windows

Therefore, for the single-window case, the traffic generated is almost the same independently of whether all local simple variables are allocated to registers or only register variables are, and it is worse if no local variables are allocated (from 29% to 34%). This indicates that a single-window architecture should have at least the register variables defined by the programmer allocated to processor registers.

The use of registers is even more advantageous if we consider the reduction in instruction memory traffic due to shorter addresses. For instance, for VAX-11 if a local simple variable is allocated to a register, then a one-byte-operand specifier is needed for a register address; otherwise, a two-byte-operand specifier is needed for a stack address.

However, the compiler should not rely on the register variable definitions done by the programmer to perform allocation because application programmers do not use them (see Section 2.2.2) and we believe that the compiler, not the programmer, should be responsible for the generation of efficient code. Therefore, we conclude that, for the assumptions made, registers should be used to store local

simple variables to reduce memory traffic, although the total data memory traffic generated is almost the same independently of whether all local simple variables are allocated to registers or only register variables are.

2.5 Register Allocation Policies

Register allocation decides which variables in a program should reside in registers. Register allocation is one of the most effective ways to optimize program execution, but it is also one of the difficult tasks performed by the compiler. For instance, Beatty [BEAT78] has modified the register allocation policy of the IBM/360 FORTRAN H compiler. The algorithm consists roughly of 4750 lines of PL/I code, i.e., it becomes a significant part of the compiler. The results were very good: the code generated by several programs is 25% smaller than the one produced by the standard compiler. No results are reported about dynamic execution, but he stated that "a comparable improvement in execution time can be expected." Chow and Hennessy [CHOW84] have measured the effect of register allocation and program optimization on the running time of 13 benchmark programs. When all the optimizations are performed, the benchmarks running time is on the average 39% smaller; however, when all the optimizations are performed except register allocation, then the running time is only 13% smaller.

This section does not pretend to be an overview of different register allocation policies —the reader is referred to [AHO78] and [LEVE83]—, but it wants to contribute in comparing two different allocation policies because as Wulf has stated "while there is data on the use of instruction sets, the relation of this data to compiler design is lacking" [WULF81]. Thus, this section discusses two different allocation policies implemented in two different compilers (the Portable C Compiler and the VMS C Compiler) for the same machine (VAX-11); the same allocation policy (in the Portable C Compiler) applied to two different machines (VAX-11 and RISC); and, finally, it presents actual research in register allocation to optimize register usage by allowing that local simple variables with disjoint lifetimes share the same register.

Two Different Compilers for the Same Machine

The Portable C Compiler [JOHN79] uses an *on-the-fly* register allocation policy: as soon as a name declaration is found the compiler decides what its storage class (register or stack) is depending only on the definition given by the programmer (register or automatic). The advantage is that the storage class is totally decided in the first pass so that the second pass can generate code once the statement has been parsed. Thus, the first pass generates the parser tree for a statement and, after that, the second pass generates code for it so that first and second pass are applied per each statement basis instead of per function basis. It results in smaller memory requirements, very convenient if the compiler has to be executed in machines with a small address space, like PDP-11.

The drawback is that registers are used only for the local variables that the programmer has explicitly defined as register (the registers are also used to store environment information and temporary results in expression evaluation; this is discussed later). Therefore, the compiler is simpler, but the generated code is far from optimal. This allocation policy corresponds to the second solution given to the alias problem in Section 2.3.

Since the compiler generates code as soon as the expression is parsed, it does not perform complex optimizations such as common expression reduction or loop invariant removal. The only optimization that is performed by the compiler is expression optimization, e.g., integer or float constant expressions are computed and operator associativity is applied so that the expression is evaluated with the minimum number of registers [SETH70].

The VMS C Compiler [ANKL82] does not allocate any storage class when a new definition is parsed. The storage class is not decided until the very end of the compilation process. Once the parsed tree has been built the Optimizer eliminates from the candidates for storage in registers those variables that the address operator is applied to them. Only the ones that cannot be referenced by any pointer variable are candidates to be stored in registers because they do not have any alias. The Optimizer also performs classical global optimizations: removal of invariant expressions from loops, elimination of common subexpression, etc.

After the Optimizer, the Local Code Generator selects the intermediate results to be hold in registers. For example, if an temporary result has to be used as an index register, then the Local Code Generator will give it the maximum priority to be allocated to a register to avoid the load instruction to bring the temporary result from memory to a register.

It is not until after this phase that the Register Allocator decides which of the variables and intermediate results are going to be allocated to registers. The Register Allocator at this moment can make a decision closer to what an assembly language programmer could do because more information is available on the candidates: usage span, loop depth, priority, etc.

Levy and Clark [LEVY82] have measured the differences in performance between both compilers executing four benchmark programs (search, sieve, puzzle, and acker). On the average, the execution time for the programs compiled using the VMS C Compiler is 2.3 times shorter than the same programs compiled using the Portable C Compiler. How much of this gain is a result of the register allocation scheme is not discussed, but probably it must be one of the most important factors.

The effect of optimization on six unspecified benchmarks is given in [ANKL82] for the VMS C Compiler. Table 2.11 reproduces the results; the first row shows the gain in execution time when all optimizations are performed and the second row shows the gain when all optimizations are performed except local variable allocation in registers. In two of the benchmarks (1 and 3) register allocation is crucial to obtain a good optimization. For instance, the running time of benchmark 1 is about 80% shorter when all optimizations are performed; if there is no assignment of local variables to registers, but all other optimizations are performed, then the running time is only reduced 20%. In the other benchmarks, register allocation helps to improve running time, but it is not as significant as those two.

	1	2	3	4	5	6
With all optimizations	22	32	15	5	68	52
Without local variables in registers	80	38	81	7	72	59

Table 2.11. Effect of Optimization on the VMS C Compiler

The Same Compiler for Different Machines

The problem with the on-the-fly allocation policy is that the code is generated per statement basis so that automatic local simple variables cannot be allocated in registers because of the alias problem. The compiler only performs local register allocation (i.e., per statement) moving a copy of the variable into a register. Thus, the compiler register allocation policy relies on the register definitions done by the programmer to perform global optimization (i.e., per block or function) so that variables are allocated directly to registers. In Section 2.2.2 we have already said that the compiler should be responsible for generating an efficient code independently of the definitions done by the programmer. Furthermore, applications programmers do not use register definitions extensively.

However, if the registers are memory mapped, then the compiler can also allocate to registers automatic simple variables. When the address operator is applied to a local variable stored in a register, its memory mapped address is computed so that both the memory address and the register number refer to the same variable. Thus, all local simple variables are allocated to registers while there are free registers. This allocation policy correspond to the fourth solution given to the alias problem in Section 2.3.

Let us see what happens when the Portable C Compiler generates code for a processor with memory mapped registers, RISC, and with unmapped registers, VAX-11. Looking again at Table 2.1 it can be concluded that:

1) The static measurements show that

1a) In RISC, all local simple variables defined by the programmer, i.e., 38% of the data objects, are allocated to registers. These objects can be addressed by only five bits.

1b) In VAX-11, half of the local simple variables defined by the programmer, i.e., 19% of the data objects, are allocated to registers, and the other half (19%) are allocated to the stack. A register address is specified by four bits for the addressing mode and four bits for the register number and a stack-frame address is specified by four bits for the addressing mode, four bits for the register number (frame pointer) and eight bits for the displacement.

Therefore, the number of bits required to refer local simple variables is significantly greater in VAX-11 than RISC. Of course, this does not imply that RISC programs are smaller than VAX-11 programs because there exist other factors to consider. In fact, the opposite is usually true [PATT82]. However, this can explain why some of the benchmark programs measured in [PATT82] have a larger program size for VAX-11 than RISC.

2) The dynamic measurements show that

2a) In RISC, all local simple variables defined by the programmer, i.e., 50% of the data objects, are found in registers: 40% are directly operated in registers and 10% are used to compute the address of the data object.

2b) In VAX-11, 38% of the data objects are found in registers and 12% in the stack frame.

Let us remember that the static measurements include the four programs, but the dynamic ones do not include the CIFPLOT program. For this reason the results are not more biased.

If no register definitions were used at all, then 50% of the data objects in RISC would still be found in registers but in VAX-11 all would be found in the stack frame.

Thus, the simplicity of the on-the-fly allocation policy does not imply inefficiency when registers are mapped and there exists enough registers to hold the whole set of local simple variables.

Register Allocation via Coloring

In Section 2.2.1 we have seen that local simple variables are the most frequent variables referenced by the programmer. Thus, the compiler has to allocate as many local simple variables as possible in registers to optimize program execution; however, the compiler must ensure that the data memory traffic reduction obtained haing these variables in registers compensate their associate register saving and restoring overhead.

To be able to allocate as many local simple variables as possible in registers, Chaitin has implemented a register allocation policy via coloring [CHAI81, CHAI82] for the IMB 801 PL.8 Compiler [AUSL82]. The lifetimes of the local simple variables over an entire function are represented in a graph such that two nodes are adjacent if they are alive simultaneously. The register allocator tries to color the graph with as many colors as available registers. Thus, variables in a function that are alive at different points during program execution can share the same color (register).

The problem of finding a minimal coloring is NP complete. Chaitin's algorithm works better when the number of available registers is high so that no spilling (i.e., transfer a variable from a register to memory) is necessary. The IBM 801 processor [RADI82] for which the algorithm was developed, has 32 registers available and, therefore, the algorithm is executed in a linear time because spilling seldom occurs. Approximate solutions have been developed [CHOW84, MCKU84] to find a non-optimal solution in a linear time for machines with smaller register sets. For instance, MIPS [PRZY84] has only 16 general-purpose registers and uses [CHOW84] algorithm.

In conclusion, actual compiler technology can solve the alias problem either with or without architectural support and can optimize register usage independently of the register definitions given by the programmer. The number of local simple variables to which the address operator is applied is very small (0.2%; see Section 2.3); thus, the number of local simple variables that are not able to be allocated to registers is insignificant. Therefore, if an optimizing compiler is available, then register usage can be almost equally efficient in a processor without memory mapped registers than in a processor with memory mapped registers.

2.6 Registers for Temporary Results

Expression evaluation is one of the most frequent constructs in C: the measured programs give that on the average 91% of the statements require an expression evaluation. There has been in the past some controversy about what kind of architecture is the best to evaluate high-level language expressions: stack-oriented, register-oriented, accumulator/stack-oriented, or storage-oriented [MYER77, KEED78, MYER78, KEED78a, KEED79, KEED83]. Recent architectures exist for both approaches: towards a stack-oriented instruction format

[DITZ82, JOHN82, BEST82, SCHU84, OHRA84] or towards a register-oriented instruction format [PATT82, BASA83, PTC83, MACG84, OLLE85]. This section discusses first the advantages of using registers to perform expression evaluation and after this, the number of registers required to store temporary results.

Registers versus Stack for Expression Evaluation

Two main reasons are usually given to justify the implementation of a stack architecture to evaluate expressions:

1) Code generated for stack architectures is more compact because of the zero-address instructions.

2) Once an expression has been parsed it is simpler by the compiler to generate code to evaluate the expression using a stack.

Myers [MYER82] has already shown that, for expressions with a small number of operators, the first statement is false, i.e., fewer instructions and smaller code are required to represent simple expressions in a register architecture than in a stack architecture. The measurements taken show that on the average there are 1.85 objects and 0.80 operators per compiled expression and 2.17 objects and 1.37 operators per executed expression. These results are similar to the ones that have already been published for other high-level languages. For instance,

(1) Alexander and Wortman [ALEX75] studied the characteristics of 19 XPL programs written by undergraduate and graduate students as well as system programmers. On the average, there are 0.76 operators per compiled expression; arithmetic expressions contain 0.41 arithmetic operators and logical expressions contain 0.28 logical operators and 0.91 relational operators.

(2) Tanenbaum [TANE78] has measured the characteristics of more than 300 procedures written in SAL used in various system programs. On the average there are 0.45 arithmetic operators per compiled expression and 1.22 operators per compiled conditional expression.

Furthermore, the distribution of the number of operators per assignment statement has also been given. Assignment statements without any operator in the right side account for 66% of the assignment statements and with only one operator account for 20% of them; only 0.3% have more than 4 operators.

(3) Cook and Lee [COOK82] have also measured statically the distribution of assignment statements in more than 120,000 lines of PASCAL programs, written by graduate students and faculty. Assignment statements without any operator account for 80% of the assignment statements and with only one operator account for 11% of them; only 0.3% have more than 3 operators.

Therefore, for the measurements shown the first statement is false. The second statement is true, but if a register allocator algorithm is already required for local simple variables (and Section 1.4 has already shown the advantages of having locals in registers instead of a stack), then the complexity added to evaluate expressions in registers is insignificant.

In conclusion, the complexity of the expressions in high-level languages does not justify the usage of special instructions to evaluate them. The compiler can use register or memory variables to perform expression evaluation. Instructions to manipulate registers are already available in the instruction set because in Section 1.4 we said that we prefer to store local simple variables in registers instead of a cache (stack). Moreover, observe that registers for temporary results do not generate any register saving and restoring overhead because they do not have to be saved across function calls*. If some registers are available to store temporary results, then no new instructions are required. The number of registers required for this purpose is discussed next.

Registers Required for Expression Evaluation

The number of registers available for expression evaluation varies with different architectures. If only an accumulator register is available and if there are parenthesized subexpressions with different operator precedences, then the accumulator has to be saved and restored during expression evaluation. So, the question is how many temporary registers are required to have a small saving and restoring overhead. We said a small overhead and not a null overhead because we want to balance the cost of the implementation (given by the number of registers available; i.e., to have a small number of registers) with the data memory traffic generated (given by the number of registers that have to be saved/restored to/from memory because no enough registers are available; i.e., to have a large number of registers)

Schneider [SCHN71] has shown that the maximum number of registers needed by a non-optimizing compiler to evaluate expressions is $(K+1)N+1$ for parenthesized expressions, where K is the maximum number of nested parenthetical subexpressions, and N is the number of precedence levels for dyadic operators. Since C [KERN78] has 16 dyadic operators in 8 precedence levels (logical operators are not included because they do not generate code for expression evaluation, but branch instructions), at most 9 temporary registers are needed for non-parenthesized expressions. However, this number is seldom used because the expressions most used by the programmer are simple ones as we have seen above. The same is true for parenthesized expressions.

*This is not true when an expression has a function call inside. In this case, temporary results have to be saved. This is discussed at the end of this section.

Since the average number of operators per expression is low (1.37 per executed expression), the number of temporary registers required to evaluate them is small. If we assume that the result given for SAL and PASCAL would be similar for C, then we can conclude that two temporary registers are enough to evaluate almost all expressions. This has been verified by two more studies:

(1) Gajewska [GAJE75] has measured statically 35,000 lines of C code written by system programmers. She has modified the Sethi-Ullman algorithm [SETH70] to compute the average number of registers required for expression evaluation on PDP-11: 1.65 for conditional expressions and 1.46 for expression statements.

(2) McKusick [MCKU84] has measured the entire set of program utilities provided by the 4.2bsd UNIX system. 97% of the expression statements and 93% of the expressions in other statements require less than 2 temporary registers to be evaluated.

Therefore, two registers are enough to store temporary results to evaluate more than 95% of the C compiled expressions. The 95% has been deduced from the numbers given by McKusick (97% and 93%) since our measurements have given that 58% of the compiled statements are expression statements.

Temporary variables are good candidates to be allocated to registers because:

a) They do not have any alias problem since the programmer cannot refer to them.

b) They do not usually have to be saved across function calls. The only exception in which registers for temporary results must be saved is is when the expression has a function call inside. In the first case, an optimizing compiler should apply associative and commutative rules to the operators in the expression so that the function is evaluated first and no temporary registers have to be saved.

c) They are only alive during statement execution. Thus, registers for temporary results can be easily shared.

If more variables for temporary results are required than registers available, then the compiler has to either save the current temporary registers in the stack or use some of the registers for local simple variables defined by the programmer. The Register Allocator knows the usage and the period of life for all local variables and tries to allocate as many as possible in registers.

2.7 Registers for Optimizing Variables

The compiler can generate some local simple variables to optimize program execution. These variables avoid that the same computation be done more than once. To decide whether to generate a local variable or to perform the same computation more than once, the compiler should not only know how many times the computation is done, but also where this computation is done, i.e., computations performed inside a loop should have higher priority than computation performed outside. Some examples of local variables generated by an optimizing compiler are:

(a) Addresses of an element of a unidimensional or a multi-dimensional array. For example, the following program

```
for (i = 0; i < MAXROW; i++)
     for (j = 0; j < MAXCOL; j++)
          total += a[i][j];
```

can be converted by an optimizing compiler to

```
for (i = 0; i < MAXROW; i++) {
     register int *p = a[i];
     for (j = 0; j < MAXCOL; j++)
          total += p[j];
}
```

so that the address computation for *a[i]* has to be done only once for each value of *i*.

(b) Common sub-expressions. For example, the following function

```
insertProc (tp, p)
proc_t   **tp, *p;
{
     if (*tp == NULL)
          *tp = p→p_next = p;
     else {
          p→p_next = (*tp)→p_next;
          *tp = (*tp)→p_next = p;
     }
}
```

can be converted by an optimizing compiler to

```
insertProc (tp, p)
register proc_t   **tp, *p;
{
    register proc_t   *q = *tp;
    if (q == NULL)
        *tp = p→p_next = p;
    else {
        p→p_next = q→p_next;
        *tp = q→p_next = p;
    }
}
```

thus, the address of the structure is kept in a register instead of following the tail pointer *tp* address specification. This causes two memory accesses for fetching the structure address.

(c) Addresses of global variables. For example, the RISC portable C compiler generates for the assignment statement:

$$c = *{+}{+}gp;$$

where *c* is a local integer variable and *gp* is a pointer to a global array of characters, the following code:

```
ldhi   #va_high,r4     load bits <31:13> of gp memory address
ldl    va_low(r4)      compute memory address and load gp
add    r5,#1,r5        increment pointer
ldhi   #va_high,r4     load bits <31:13> of gp memory address
stl    r5,va_low(r4)   compute memory address and store gp
ldbs   0(r5),r30       load operand in c
```

An optimized compiler would detect that the same address is needed and *r4* would be loaded only once.

These compiler defined variables can be kept in registers. The Register Allocator has to weigh their usage and priority against the usage and priority of local simple variables defined by the programmer and to decide which ones are going to be allocated in registers. In addition to their usage and priority, the Allocator must also know their period of life to try to share the smaller possible number of registers. If no function calls are involved, then temporary registers can be used, i.e., registers that are not automatically saved across function calls; otherwise, registers for local simple variables saved across function calls should be used.

The C compiler that has been used to take the measurements does not perform these optimizations because it generates code as soon as the statement has been parsed. So, no measurements were taken about how many registers are required to store compiler defined variables. Even though there exists abundant information about optimization, no information has been found concerning either how many temporary variables a compiler generates or how many registers are needed for compiler optimization. Therefore, we must leave this number as undefined, although we do not expect that the number of registers required by the compiler would be more than two. The reason is that, on the average, the number of **for**, **while**, and **do** statements parsed per function is small (0.59 per function), and so is the average number for unidimensional arrays (2.8% of the data objects). The usage of multi-dimensional arrays for the measured programs is negligible (0.003% of the data objects). Furthermore, the more optimizing variables are required in a function, the higher the probability is that their lifetimes be disjoint so that they can share the same register.

If there are more optimizing variables than registers, then the compiler has to either define some local simple variables or use some of the registers for local simple variables defined by the programmer. As we said for temporary variables the Register Allocator knows the usage and the lifetime for all local variables and tries to allocate as many as possible in registers. However, if there are no more registers available, then it could be better to perform the same computation more than once rather than to have to create an activation record in the stack for only optimizing variables (assuming that this is not required for local variables defined by the programmer).

2.8 Registers for Parameter Passing

This section discusses how the compiler can pass parameters from the caller to the callee. When the compiler is parsing a function call statement, it does not have any information about what is the usage of the parameters for the callee; therefore, three alternatives are possible:

(1) To pass parameters through the stack. When the compiler parses a function definition, it has full information about the parameter usage and can decide which parameters should be moved into registers and which ones should remain in the stack frame. Again the programmer can also use register variable definitions to give a hint to a non-optimizing compiler.

(2) To pass parameters through registers. The compiler usually divides the register set into registers for temporary results and registers for local variables. The first ones are not saved across function calls and the second ones must be saved either by explicit or implicit instructions, or with a multiple window scheme. The compiler can use the temporary registers to pass the parameters. The compiler has again full information about their usage so that it can decide which ones are moved to a register for local variables and which ones are moved

to the stack frame. If there are more parameters than registers, then only the first ones are passed through registers and the others through the stack. If multiple non-overlapping windows are available and the register set has some registers common to all the functions (i.e., not saved across function calls), then these registers can be used to pass the parameters.

The drawback of this approach is that once a parameter has been computed, there is one less temporary register available to perform expression evaluation. However, the number of parameters per function call is very low: the static measurements give on the average 1.55 parameters per function call and for the dynamic measurements this number is reduced to 0.99 parameters per function call. Furthermore, we have already mentioned that in C the most frequent expressions are the simple ones (see Section 2.6); this is also true for parameter expressions. Table 2.12 shows the parameter types that are passed to a function: 12% are constants, 56% are simple variables, and only 32% are either an expression to be evaluated or a non-simple variable (an element of an array or a field of a structure). Since C does not require to evaluate the parameters in any specific order, the compiler can reorganize the parameter expressions so that the ones that require more temporary registers to be evaluated, are evaluated first. Thus, temporary registers can be used to pass parameters to avoid data memory traffic.

parameter type	NROFF no.	NROFF %	SORT no.	SORT %	VPCC no.	VPCC %	ALL PROG no.	ALL PROG %
constant	122550	7.6	17	-	650786	14.1	773353	12.1
global sv.	671705	41.4	50	-	14526	0.3	686281	10.7
auto sv.	133881	8.3	2404	1.3	979340	21.3	1115625	17.4
register sv.	508309	31.3	55417	28.9	1225843	26.6	1789569	28.0
complex	185907	11.5	113777	69.8	1734581	37.7	2034265	31.8
TOTAL	1622352		171665		4605076		6399093	
param. per ftn.	0.49		1.21		1.51		0.99	

Table 2.12. Function Parameters

(3) To pass parameters through registers in multiple overlapped windows. In this case, the window is divided into three parts: registers for outgoing parameters, registers for incoming parameters, and registers for local variables as it is discussed in [PATT82]. The overlapped registers become part of the local registers of the current window so that the compiler does not have to move them either to local registers or to the stack. If the number of overlapped registers is smaller than the number of parameters, then the first ones are passed through registers and the others through the stack; the compiler has again full information and can decide whether to move any of the parameters in the stack to registers and vice-versa. If the number of parameters is smaller than the number of overlapped registers, then no memory traffic is produced because a new window of registers is available for each function call.

Traditionally compilers have used the first approach for parameter passing in architectures without multiple windows, but research for optimizing compilers is moving toward the second approach to reduce data memory traffic overhead caused by parameter passing [CHAI81].

Looking again at Table 2.10, we can deduce that the overhead caused by parameter passing with respect to the data memory traffic generated by the data objects is 10% in the case that only explicitly defined register variables are allocated to processor registers (SW (a)), 13% in the case that all local simple variables defined by the programmer are allocated to registers (SW (b)), and 5% in the case that no local simple variables are allocated to registers (SW (c)). If parameters are passed through registers and enough registers are available, this overhead is eliminated.

number of parameters	% of functions with up to n parameters								
	static					dynamic			
	CIFPLOT	NROFF	SORT	VPCC	A.P.	NROFF	SORT	VPCC	A.P.
0	30.7	65.5	28.6	15.1	35.7	57.4	0.0	10.4	34.9
1	59.9	92.0	57.2	66.7	71.4	94.1	79.5	60.1	78.4
2	82.9	99.1	95.3	88.5	89.8	100.0	100.0	91.8	96.2
3	91.7	99.5	100.0	96.0	95.6			96.0	98.1
4	95.0	100.0		99.2	97.9			96.6	98.4
5	96.5			99.6	98.5			96.8	98.5
6	96.9			100.0	98.8			100.0	100.0

Table 2.13. Registers for Parameter Passing

Table 2.13 shows the percentage of functions that pass n or less parameters for both static and dynamic measurements. Our first conclusion from the table is that functions with a small number of parameters (≤ 3) are called more frequently that functions with a big number (> 4). For the three measured programs, three registers are enough to pass parameters for 98% of the function calls. If four registers are available a gain of 0.3% is obtained and with registers the gain is 0.4%. To be able to have all parameters passed through registers, six registers are required.

In conclusion, to reduce the overhead caused by parameter passing, parameters can be passed through registers: temporary registers for single windows or overlapped registers for multiple windows. In the former case, the compiler should decide to move the parameters either to the stack or to registers for local variables; thus, some extra instruction memory traffic is generated for instructions to transfer parameters from temporary register to registers for local simple variables or to the stack. Since C does not require to evaluate the parameters in any specific order, the compiler can reorganize the parameter expressions so that the ones that require more temporary registers to be evaluated, are evaluated first.

The number of registers required is not very large: three registers are enough for 98% of the functions.

Finally, before leaving this section we must considered one of the features of C that makes impossible to pass parameters through registers, except if its usage is restricted. The C programmer can pass a variable number of parameters to the callee. This is very useful, for instance, in the *printf* function [KERN78]. The caller can specify as many parameters as necessary to be printed; the callee uses the first parameter to control the printing output format and to know how many parameters have been passed. In this case, the programmer of the callee function only specifies one parameter and uses the address of this parameter as the address of an array of parameters. Of course, the resulting program is not a machine independent program, although the programmer has available a include file (*varargs.h*) to make this mechanism as general as possible. This feature is a consequence of the flexible type manipulation (using the address of a parameter as the address of an array) and the lack of type checking provided by C. Languages with a strong type checking mechanism (PASCAL, ADA) do not allow to pass a variable number of parameters.

Since we are passing the first parameters through registers, then we cannot use an array to refer them. Therefore, the compiler cannot pass parameters through registers because it does know if the callee refers them as members of an array. They are two possible solutions to this problem:

(1) The programmer should specify a special type definition for the functions that expect to use the list of parameters as an array.

(2) The programmer should specify a special type definition to define a variable list of parameters. Every time that the function is called the programmer should *cast* the list of parameters with this definition type to indicate to the compiler that the callee expects a variable number of parameters.

In any case, the compiler knows that a function (in both caller and callee level) is processing a variable number of parameters so that it might decide to pass all of them through the stack or the first ones through registers and the rest through the stack (if any). Although no measurements are available, we would affirm that this feature is seldom used for either application or system programmers. Therefore, the definition of C should be modified as indicated above to allow the compiler to pass parameters through registers.

2.9 Environment Registers

This section discusses the number of registers required to store the run-time process environment. The run-time process environment can be divided into two different categories: privileged and general.

The privileged process execution environment is defined by the processor status: processor mode (user or supervisor), processor priority, interrupt mask, etc. It also includes registers associated with memory management information to support virtual memory. Privileged registers are not discussed in this thesis.

The general process execution environment is given by the following four elements:

Program Counter Points to the next instruction to execute.

Stack Pointer Points to the top element in the execution stack.

Frame Pointer Points to the current activation record in the stack.

Argument Pointer Points to a list of parameters being passed to a function.

This does not mean that all four need to have a register associated to it. Section 2.9.1 shows that the frame pointer and the argument pointer can be only one register for C programs and Section 2.9.2 shows that the frame pointer and the stack pointer can also be only one register for C programs if an optimizing compiler is available.

Some other environment registers might be required by the architecture. For instance, if the architecture has multiple windows, then two more registers are required:

Current-Window Pointer Contains the memory address for the current window in the register file.

First-Window Pointer Contains the memory address for the first window available in the register file.

The usage of these two registers is discussed in Chapters 3 and 4.

2.9.1 Using the Frame Pointer as Argument Pointer

This subsection shows how the frame pointer can also be used as argument pointer when parameters are pushed from right to left (i.e., the rightmost parameter pushed first and the leftmost parameter pushed last) and an optimizing compiler is available. To simplify our discussion let us assume that the whole activa-

tion record is in the stack and no multiple windows are available. Note that otherwise the discussion should be applied only to the part of the activation record which resides in the stack.

If the parameters are pushed from left to right, then the argument pointer is required to indicate to the callee where the first (the leftmost) parameter is stored. Since C allows to pass a variable number of parameters (see Section 2.8), the callee does not know where the first parameter is stored because the stack pointer points to the last element pushed (to be precise, to the return address after the call, but it would be possible to find the last element). Figure 2.3 shows the activation record that the function has to set before executing: the argument pointer points to the first parameter, the frame pointer to the beginning of the locals, and the stack pointer to the top of the stack; the return address, the old frame pointer, and the registers have been saved between the parameters and the locals. Note that the parameter pointer has to be explicitly initialized by the caller, saving its old value to be restored when the callee returns. The frame pointer is loaded with the value of the stack pointer after its contents and the contents of the general-purpose registers have been saved. The stack pointer is decremented (if the stack grows downwards) to reserve space for the local variables defined by both the programmer and the compiler. These operations can be performed with one or several instructions depending on the architecture; this is not discussed here because it does not have any influence in deciding the number of registers required.

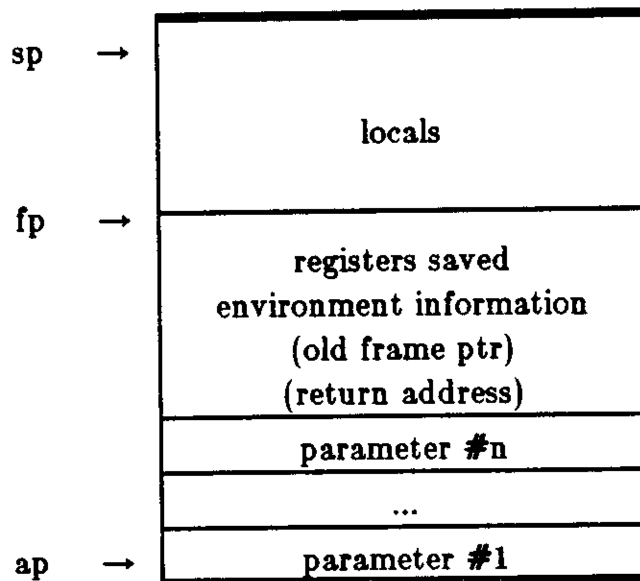


Figure 2.3. Activation Record Using Frame, Argument, and Stack Pointers

On the other hand, if the parameters are pushed from right to left so that the first parameter is always on top, then when a function is called the frame pointer is loaded with the current value of the stack pointer so that parameters are accessed with a positive displacement and locals with a negative displacement with respect to the frame pointer (assuming again that the stack grows downwards). The activation record is shown in Figure 2.4.

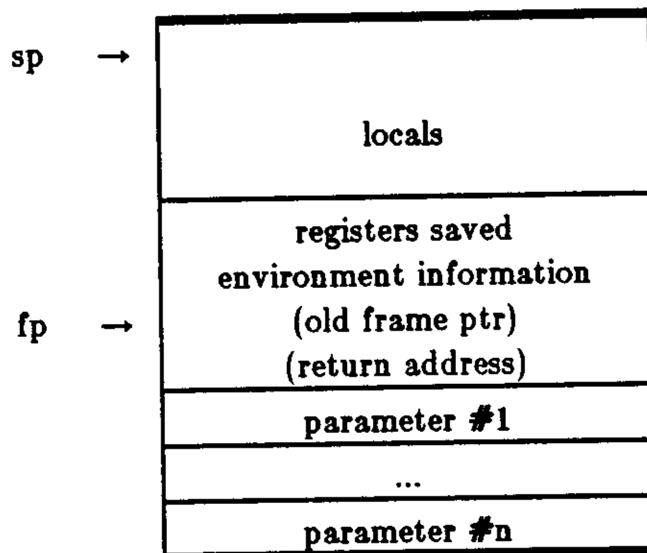


Figure 2.4. Activation Record Using Frame and Stack Pointers

However, the order in which the parameters are pushed is not enough to reduce both the argument and the frame pointer to only one register. Again we have to consider the optimization degree of the compiler. If we assume that the code is generated as soon as the statement has been parsed using an on-the-fly allocation policy (e.g., the Portable C Compiler), then the compiler has no knowledge about the number of registers used for local simple variables before starting generating code. Thus, the compiler cannot generate displacements for local simple variables in the stack because it does not know the number of register that are saved between the old frame pointer and the space for local simple variables (see Figure 2.4).

This problem can be solve either using two pointers (argument and frame) or saving a fixed number of registers. If two pointers are used, then displacements for local simple variables can be generated independently of the registers that must be saved (see Figure 2.3). If the same number of registers is used, then displacements can also be generated because there is always a fixed displacement to the first local simple variable in the stack. In this case, the number of registers saved is independent of the number of registers used to store local simple variables, i.e., the registers are always saved even though they are unused by the function.

The first solution is used by the Portable C Compiler for VAX-11. The created activation record for VAX-11 is similar to the one given in Figure 2.3. The difference is that parameters are also pushed from right to left so that the argument pointer can be initialized in the moment of performing the call instead of having to be initialized before pushing the parameters. The second solution is used by the Portable C Compiler for PDP-11. Three registers are used for register variables defined by the programmer. As we said, these registers are always saved independently of whether the function is using them. The created activation record for PDP-11 corresponds to the one given in Figure 2.4.

If an optimizing compiler is available so that code is not generated until the function has been parsed, then the number of registers used by local simple variables is know and, therefore, the same register can be used for both the argument and the frame pointer. This problem is similar to that of other block-structured languages (ALGOL, PASCAL, ADA), except that the order in which parameters are pushed is not important because no variable number of parameters is allowed.

2.9.2 Using the Stack Pointer as Frame Pointer

If an optimizing compiler is available and no code is generated until the function has been parsed, then the following data is known:

1) The number of local variables defined by the programmer.

2) The number of temporary and local variables required by the compiler.

3) The maximum number of parameters required by any of the function calls that appear in the function.

4) The number of registers that must be saved.

In this case, the size of the activation record is obtained by adding these four numbers plus one word for the return address. When the function is entered, the stack pointer is decremented by the activation record size. Displacements for local and temporary variables are computed relative to the stack pointer so that no frame pointer is necessary. When the function returns either the stack pointer can be explicitly incremented by the activation record size or implicitly restored if it was saved by the execution of the call instruction. Consequently, as Figure 2.5 shows, the stack pointer can be used as frame pointer and argument pointer. Observe that the order of evaluation of the parameters is no longer important; however, the compiler must allocate them so that the leftmost parameter is the first below the return address, the second leftmost below the first, and so on (assuming again that the stack grows downwards).

A similar scheme was proposed by Ditzel and McLellan [DITZ82] in their C Machine. The C/70 machine [BBN81] uses also a similar scheme. However, the activation record does not contain the maximum number of parameters that can be passed to the function; every time a parameter is pushed (and the stack pointer decremented) the compiler has to modify the displacements for the local variables.

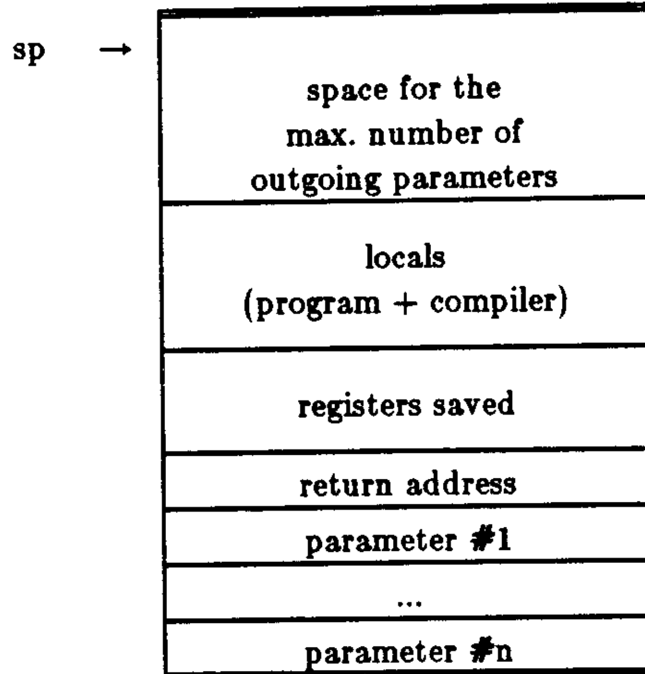


Figure 2.5. Activation Record Using Only Stack Pointer

Therefore, if an optimizing compiler is available, the stack pointer, the frame pointer, and the argument pointer can be combined in only one register.

2.9.3 Registers Required

In conclusion, for optimizing compilers only two registers are required for the general-process execution environment: the program counter and the frame or stack pointer. If multiple register-windows are implemented, two more environment registers are required as we will discuss in Chapters 3 and 4: the current-window pointer and the first-window pointer.

2.10 Registers for Global Simple Variables

This section discusses why general-purpose registers are not used to allocate global (external or static) simple variables and shows how some special registers addressed through memory addresses can be used. Looking again at Tables 2.5-2.8 we can deduce that global simple variable data objects (with and without autoincrement or autodecrement operators) are the most executed objects after

local simple variables and integer constants: on the average they account for 13% of the data objects. If local simple variables are kept in registers, then the data memory traffic caused by global simple variables accounts on the average to 44% (remember that no overhead caused by function calls was accounted on that tables yet).

If the compiler were able to allocate them in general-purpose registers (let us not worry about the number yet), then data memory traffic would be reduced. However, compilers do not store global simple variables in the general-purpose registers either permanently or temporary. Let us discuss why.

The compiler cannot decide to store a global simple variable permanently in a register because C modules are compiled separately so that no information is available to the compiler about the other modules. Thus, the compiler cannot have a global policy to allocate global simple variables to registers. The compiler could collect information in the symbol table about the usage of each global variable in the same way that it is done for local variables [FREI74]. This information is passed to the loader in the object module. However, even if the loader had complete information about the usage of each global variable, it would have to change the code generated by the compiler changing memory references to register references. This is not practical.

The compiler could decide for each function which global simple variables are moved to registers. In this case, the compiler has the same problem that we have mentioned for local variables: aliasing. The copy of a global value is valid until either the next function call or the next expression evaluation which includes a pointer indirection. As has been mentioned above, one of each 25 executed data objects is a function call and one of each 50 executed data objects is a pointer indirection. Thus, the expected period of life of a variable moved to a register is very short if we cannot guarantee that the variable has no alias.

Therefore, compilers are faced with a similar problem to the one discussed for automatic simple variables in Section 2.3. Moreover, the programmer cannot give any hint to the compiler because register definitions for global variables are not allowed in C. There are three different solutions:

(1) Global simple variables are not allocated to registers either permanently or temporary. It results in high memory traffic and slow access to the variables. The Portable C Compiler, for instance, uses this policy.

(2) Optimizing compilers can allocate global simple variables temporary to registers. However, optimizing compilers cannot applied for global simple variables a similar policy to the one used for automatic variables, i.e., the compiler allocates a global simple variable to a register if the programmer does not apply the address operator to it (see solution (3) to the alias problem in Section 2.3). The

reason is that the compiler does not know if the address operator has been applied to the external variable in another module. Therefore, the copy of the global variable is valid only until the next function call or the next pointer indirection so that the expected period of life is very short.

(3) The architecture can again offer some support to solve the alias problem so that the compiler can allocate permanently some global simple variables to registers. The solution is to have some memory locations mapped to registers. The compiler keeps similar information in the symbol table to the one kept by automatic simple variable to decide which ones are the best candidates to be allocated in registers (static frequency usage with a relative weight depending on whether the variable is used inside a loop statement). The loader when decides the memory addresses for global variables using the information left by the compiler in the object modules, then it also decides which ones are stored in these memory locations, i.e., allocated to registers. Note that the loader does not have to change any code generate by the compiler because these registers are not referenced by a register address, but by a memory address.

program	no. scalar global var.	90% ref. occurs in
pcc	95	16
troff	234	46
dr	79	18

Table 2.14. Registers for Global Simple Variables

How many memory locations are needed for this purpose? Ditzel and McLellan [DITZ82] propose to have 16 special memory locations for global simple variables. This number is based on their measurements reproduced in Table 2.14. The measured programs are: the Portable C Compiler (*pcc*) compiling the machine dependent part of the compiler, the UNIX word processor *troff* processing the troff tutorial, and a VLSI design rule checker (*dr*) checking a 32-bit adder.

McDaniel [MCDA82] proposes to have only four memory locations for global simple variables. He has measured two programs: the MESA compiler and a VLSI circuit analysis program. His measurements show that 24% and 17% of the instructions of these two programs are converted from memory instructions to register instructions when four registers are available to map global simple variables and four registers are available to map local simple variables (MESA is a stack architecture so that all local simple variables are allocated to memory). Since the measurements are given for both optimizations we cannot conclude how much memory traffic reduction is obtained with only four mapped memory locations for global simple variables.

In conclusion, global simple variable memory traffic can be reduced if the architecture offers some registers mapped to special memory locations. The compiler has to keep usage information about global simple variables similarly to the information kept for automatic simple variables to decide register allocation; this information is passed to the loader through the object module. The loader looks for the best variable candidates to be in registers and allocates them to these special memory locations. No measurements were taken in this report regarding how many registers are needed for global simple variables; the measurements taken by Ditzel and McLellan are valid and, therefore, their proposal of having sixteen registers for global simple variables is adopted.

2.11 Conclusions

In this chapter we have studied the usage of registers for different purposes:

(1) Registers for local simple variables defined by the programmer. We have shown that they account for half of the data objects that the measured programs manipulate. This allow us to conclude that the compiler should allocate these variables to registers independently of the storage class specified by the programmer, automatic or register, to reduce both the program size and the data memory traffic generated during program execution. Although a single window is available, local simple variables should be allocated to registers because (a) register saving and restoring overhead is balanced with the reduction in data memory traffic generated by the data objects, and (b) shorter addresses can be used so that the instruction memory traffic is also reduced. The efficiency of the generated code should depend on the optimizations done by the compiler, not on the storage class definitions done by the programmer.

The alias problem can be easily solved by an optimizing compiler without having any architectural support, i.e., without having memory mapped registers. The number of local simple variables that are not able to be allocated to registers by an optimizing compiler because the address operator has been applied to them is very small: only 0.2% of the data objects are local simple variables with the address operator and they account for 0.0013% of the executed objects. Thus, register usage can be almost equally efficient in a processor without memory mapped registers than in a processor with memory mapped registers if an optimizing compiler is used.

The number of local simple variables defined by the programmer for each function is not very large: 99% of the functions have up to fourteen local simple variables. However, we have seen that the most frequent called functions are the ones with a small number of local simple variables: 75% of the called functions have up to three local simple variables, including parameters passed to the function. The number of registers necessary to allocate local simple variables depends on the register allocation policy. If a register allocation by coloring is used, then

some local simple variables can share the same register if they have disjoint lifetimes.

(2) Registers for temporary results and parameter passing. The number of registers necessary to perform expression evaluation is also not very large: two registers are enough to evaluate more than 95% of the expressions. Registers for temporary results can also be used for parameter passing for single windows. In this case, three registers are enough to pass parameters for 98% of the functions.

(3) Registers for optimizing variables. No measurements have been taken nor found to determine the number of registers required by a compiler to store optimizing variables. However, we do not expect to use more than two registers for each function.

(4) Registers for the run-time process environment state. If an optimizing compiler is available, then only two registers are required: the program counter and the frame pointer (also used as stack pointer and argument pointer). If multiple windows are available, then two more registers are required: the current-window pointer and the first-window pointer.

(5) Registers for global simple variables. Although global simple variables cannot be allocated to general-purpose registers, they can be allocated to some memory locations mapped to special registers so that they can only be addressed through memory addresses. In this case, only the data memory traffic is reduced, not the instruction memory traffic.

Therefore, if an optimizing compiler is available, registers can be used very effectively to reduce both the data memory traffic and the instruction memory traffic, even when only a single window is available. Our measurements indicate that if only a single window is available, then the number of registers available in the general-purpose register set should be either 32 or 16 plus some specialized registers (the program counter, for instance). Our measurements do not justify the actual trend of some architectures oriented to UNIX (i.e., C) that provide more than 32 general-purpose registers (PYRAMID 90x and CELERITY C1200).

In the next chapter we introduce how the register saving and restoring overhead can be reduced using multiple windows, and we study how the register set should be organized. We also consider the number of registers that should be available for each purpose to balance the implementation cost of the register file and the number of bits required in the instruction to specify a register address with the number of variables defined by both the programmer and the compiler and the data memory traffic overhead introduced by register saving and restoring (or overflows and underflows in the case of multiple windows).

CHAPTER 3
MULTIPLE WINDOWS

3.1 Introduction

This chapter studies the architecture of a register file for a processor oriented to execute programs written in C. The design is based on the measurements presented in the previous chapters. The proposed register file is compared with those of other recent proposals known by the author: C/70 [KRAL80], RISC [PATT82], PYRAMID 90x [PTC83], and CELERITY C1200 [OLLE85].

Multiple windows allow to reduce the register saving and restoring overhead as has already been shown by Patterson and Séquin [PATT82]. When a function is called, a new window is allocated from the register file. So, no register saving and restoring is involved if enough windows are available in the register file. Windows can overlap to pass parameters through registers to reduce the parameter passing overhead as has been discussed in Section 2.8. RISC, PYRAMID 90x, and CELERITY C1200 have overlapped windows (six registers for RISC and sixteen for PYRAMID and CELERITY). When no more windows are available, then a register file overflow exception is generated. These processors handle overflow in three different manners:

(1) To have enough windows in the register file so that an overflow exception is never generated. This is implemented by C/70 with a register file of 128 windows.

(2) To allocate *virtual windows* from memory when there are no more windows available in the register file. This solution is implemented by CELERITY that has sixteen windows per register file and eight register files available for eight active processes (each file is private to each process; see Section 3.9.2). Therefore, processes with a window stack depth larger than sixteen are not using registers, but memory locations referenced as registers.

(3) To remove the oldest window in the register file, i.e., to make space for the new one. This is implemented by RISC and PYRAMID 90x. In both processors the register file is organized as a circular buffer to avoid to relocate the windows when an overflow exception is detected. RISC has a register file of eight windows and PYRAMID of sixteen. However, since the windows overlapped to pass parameters up to seven windows for RISC and fifty for PYRAMID can be active at any moment (see Section 3.5).

In this chapter we consider the third approach, i.e., when an overflow occurs the oldest window is removed from the register file and saved into the *window overflow stack*. However, how the register file is implemented is not discussed until Chapter 4.

A special mention has to be done with CELERITY C1200. The C1200 has a NCR/32 processor [NCR83] as internal processor and an external 4K register file. NCR/32 is a single chip VLSI CPU processor with a cycle time of 125 ns. and only sixteen internal registers. Since instructions cannot operate directly with external registers, NCR/32 provides some special instructions to transfer an external register to an internal one. Thus, the register file is organized as a program-controlled cache of registers. Even though no information is available, we can guess that the compiler has a single-window-oriented register allocator with the register saving and restoring overhead discussed for single windows. The differences with a stack-oriented approach are that fast access to the operands stored in registers is provided (only one instruction of 125 ns.), and that short addresses are used (5 bits in a 16-bit instruction to specify the virtual register number of the current window). Therefore, C1200 is not a multiple-window architecture in the same sense than the other ones are because registers in the windows are not available to the native processor instructions. However, since no study has been made about the effectiveness of this approach and very little information has been released, in this thesis we just consider the multiple windows, i.e., the overhead caused to transfer information for the register-file cache to the internal registers is transparent to our discussion.

All the existing processors with multiple windows use fixed-size windows: C/70 has a window size of eight registers, RISC of sixteen registers*, and PYRAMID 90x and CELERITY C1200 of thirty-two registers. Since the number of registers required per function is usually small (75% of the functions have up to three local simple variables, including parameters; see Section 2.2.3), part of the registers in the window are being unused. Ditzel and McLellan [DITZ82] propose the use of variable-size windows to improve the use of registers, but this results in a complex and slower implementation (see Section 3.3). As a compromise that incorporates the advantages of both schemes, we propose an approach that uses a few (2 or 3) window sizes (Section 3.4). We use our measurements to compare with the fixed-size window approach and to determine suitable window sizes (Section 3.7). We show that for an equivalent performance to the fixed-size case the register file can be made significantly smaller (Section 3.8).

*The first implementation of RISC I had a smaller window size. Sixteen registers corresponds to the window size for the second implementation of RISC I and for RISC II that have been widely used in the literature.

This chapter is organized as follows. Section 3.2 confirms with our measurements that the use of a multiple-window register file reduces dramatically the overhead in register saving and restoring. Section 3.3 compares fixed-size windows and variable-size windows. Section 3.4 presents the an intermediate approach of using a few different sizes that offers most of the advantages of both approaches. Section 3.5 considers the advantages of having overlapped windows versus non-overlapped ones. Section 3.6 explains how the register set is organized with multi-size overlapped windows. Section 3.7 uses our measurements to select the different window sizes. Section 3.8 also uses our measurements to select the size of the register file and shows that in the proposed approach the size is significantly smaller than in the fixed-size window case. Finally, Section 3.9 studies the effects of exceptions and context switching on the register file and presents two different organizations for the register file: private register file and shared register file.

3.2 Data Memory Traffic Overhead for Multiple Windows

This section confirms that multiple windows reduce dramatically the overhead in register saving and restoring. Since the purpose of this section is not to discuss the overhead generated by different window and file sizes (this is done in Section 3.8 after we have introduced our proposal), only one specific size for the window and one for the file have been considered. Instead of taking arbitrary sizes, we have chosen the sizes available in one of the existent processors (RISC) as we will see below. A similar approach was taken to compute the overhead for single-window architectures (see Section 2.4).

For the multiple-window architecture style we assume that the window has enough registers to store all local simple variables defined by the programmer. This corresponds to case (b) for single windows (see Section 2.4). Since for a single-window architecture we have concluded that it is advantageous to use registers to allocate simple variables and a multiple-window architecture is register oriented, the register allocation policies equivalent to cases (a) and (c) are not considered here. Furthermore, we have already seen that the number of registers required for local simple variables is not very large: 99% of the functions have up to fourteen local simple variables defined by the programmer (see Section 2.2.3). The multiple window case we consider is called "MW" in the sequel and the data memory traffic is shown in Table 3.1. The total data memory traffic for the three single window cases has been copied from Table 2.10.

Since a multiple-window architecture benefits from the use of overlapped registers to pass parameters (this is done in RISC), we assume that the windows are overlapped and that they have sufficient registers to pass all parameters (the five registers in RISC are sufficient to pass parameters for 98% of the called functions) Thus, parameters are not pushed onto the stack and no data memory traffic overhead caused by parameter passing is generated for MW.

	NROFF	SORT	VPCC	ALL PROGRAMS
No. function calls	3309538	142398	3041209	6493145
SW (a) data memory traffic	64959158	7067770	71554324	143958384
SW (b) data memory traffic	64794921	4211286	79000676	149851557
SW (c) data memory traffic	83478696	8762290	100562598	192803584
MULTIPLE WINDOW				
data memory traffic	26542603	1667271	26222849	54392487
data objects	23259541	1667271	22816695	47743507
parameters	-	-	-	-
registers	3283062	-	3406154	6648980
% overhead (tot./d.obj.)	14	0	15	14
SW(a) vs. MW overhead	145	324	173	165
SW(b) vs. MW overhead	144	153	201	176
SW(c) vs. MW overhead	215	426	283	254

Table 3.1. Overhead for Multiple Windows

To compute the traffic due to overflows and underflows, we use the values from Table 3.5 for seven windows. Note that the the values reported in [PATT82] for the same number of windows is only 1%; this is because this article uses two short benchmark programs (sort and puzzle) to measure the percentage of overflows and we have already said that they are not representative enough for this type of measurements (see Section 1.3). When an overflow occurs one window must be saved in memory. We have assumed a window size of sixteen registers (like RISC's window size) to compute the overhead caused by register file overflows and underflows. The number of function calls executed for each program was given in Table 2.10 and is reproduced in Table 3.1. Thus, from Table 3.1 we deduce that MW has 14% more data memory traffic than that strictly caused by the programmer.

With respect to single windows, the data memory traffic generated for SW (a) is 165% greater than the one generated for MW, for SW (b) it is 176%, and for SW (c) it is 254%. Therefore, data memory traffic is reduced by a factor of 2.7 or 3.5 by using multiple windows.

We also observe in Table 3.1 that for the SORT program, all the three single window cases have a higher overhead with respect to multiple windows than the other programs: from 324% if only register variables are allocated to processor registers to 426% if no local simple variables are allocated to registers. This is because SORT has a higher percentage of local simple variables with or without auto-increment or auto-decrement operators (53% in comparison with 40% for NROFF and 38% for VPCC), and a lower percentage of function calls (2.1% in comparison with 4.6% for NROFF and 3.8% for VPCC).

In conclusion, when multiple windows are available, the data memory traffic is reduced by a factor of 2.7 or 3.5 with respect to the case in which a single window is available. Therefore, multiple windows offer an excellent support for function calls independently of the instruction set design (reduced or complex).

3.3 Fixed-Size Windows versus Variable-Size Windows

The number of registers allocated in a *variable-size window* is exactly the number of registers required per function. On the other hand, if *fixed-size windows* are used, then there are some functions that require more registers and some functions that do not use the whole set of registers available. Therefore, variable-size windows allow a more efficient use of the register file. This implies that the register file can be smaller because fewer registers are required to hold the same number of windows, and fewer registers have to be saved (restored) when an overflow (underflow) occurs and during context switching. However, variable-size windows have four main drawbacks:

(1) Since the purpose of variable-size windows is to have enough registers for all local simple variables, the number of bits available in the instruction to address a register is determined by the maximum number of local simple variables that can be defined in a function. This results in a larger number of bits required for a register address in the instruction format.

(2) If the window can be of any size, random access to the whole register file is required. Fixed-size windows require random access only to the current window, which permits an implementation that has a short bus whose size is independent of the register file size. If the register file is implemented as a circular buffer (like RISC and PYRAMID 90x register files), then random access to the whole register file is still required. However, an alternative implementation to the circular register file called shift register file introduced in Section 4.3 allows this independence.

(3) Physical register addresses have to be computed every time a register is accessed. This computation requires the addition of the current-window pointer with the virtual register number given by the instruction. If three-address instructions are available the overhead introduced is significant except if all three additions can be performed in parallel. Consequently, the register address computation in the fixed-size window case is certainly faster than the one in the variable-size window.

(4) When a function returns it is not possible to check if the caller window is completely present in the register file because the callee does not know the caller's window size. To solve this problem two alternatives are possible:

a) Every instruction that refers to registers in the current window checks if the register is present in the register file. If it not, then a trap is generated to bring it from memory. If only one register is brought, then underflows will be really expensive to handle. If a fixed number of registers is brought, then it is possible that those that are not part of the current window have to be saved again without having been utilized.

b) To have a special instruction in the caller placed after every call instruction (and, therefore, executed after every return instruction) that verifies that the current window is present in the register file. This causes an increase in the number of instructions required to execute a function call to four: two instructions in the caller (the call itself and the verification of the presence of the current window) and two in the callee (the allocation of the new window and the return).

Ditzel and McLellan [DITZ82] have proposed a C Machine with multiple variable-size windows. Each window correspond to an activation record for local simple variables. The register file contains the top windows in the stack. Operands are referenced with a displacement to the stack pointer which is also used as a frame pointer. To reduce the overhead of having to compute the register address every time an operand is referenced, the Instruction Cache stores partially decoded instructions, i.e., operand addresses of the form *SP + offset* are converted either to a register address, if the operand is in the register file, or to a memory address, otherwise. The presence of the current window in the register file, after a function call, is ensured by executing a special instruction when the callee returns. Thus, this machine has three of the above mentioned drawbacks (1, 2, and 4) and has reduced the third one by increasing the complexity of the processor.

In conclusion, although variable-sized windows utilize the register file more efficiently than fixed-sized windows, their implementation increases significantly the complexity of the processor: large register addresses are required in the instruction, the bus size cannot be made independent of the register file size, more instructions are required to execute a function call, and more overhead is incurred to map a virtual register address to a physical register address. Fixed-size windows are implemented in C/70, RISC, PYRAMID 90x, and CELERITY C1200. The following section shows that it is possible to have a small number of sizes (2 or 3) to balance the advantages of both approaches.

3.4 Multi-Size Windows

A second alternative to fixed-size windows, less flexible that variable-size windows, is to have available more than one size. By default, the smaller set of registers is always allocated. Only when the compiler detects that the function needs more registers, then it generates an instruction to allocate a larger size window. These windows are called *multi-size windows*.

The architecture does not have to provide any protection support to ensure that an instruction is referring to a register inside the window. This is ensured by the compiler because the programmer is not using an assembly language, but a high-level language.

Three of the advantages given for fixed-size windows are still valid for multi-size windows: small register addresses are required in the instruction, the bus size can be made independent of the register file size, and less overhead is incurred to map a virtual register address to a physical register address (in comparison with variable-size windows). In addition, the advantages given for variable-size windows are also valid for multi-size windows: registers are better utilized, i.e., the register file can be smaller because less registers are required to hold the same number of windows, and less registers have to be saved (restored) when an overflow (an underflow) occurs and during context switching.

With respect to variable-size windows, multi-size windows are more restricted because the largest window size might not be sufficient for some functions so that some local simple variables might have to be allocated to the stack, and some of the registers in the window are unused when the number of variables does not match a window size. However, in Section 3.7 we will see that an adequate choice of the sizes makes the former occur infrequently and the number of unused registers small.

With respect to fixed-size windows, multi-size windows offer two drawbacks. The first drawback is that the mapping from a virtual register address to a physical register address is faster for a power-of-two fixed-size window because no addition is required, only concatenation. However, the time and chip area required to perform the addition of a few bits is almost insignificant. This addition must also be performed for non-power-of-two fixed-size windows (see Section 4.2.4).

The second drawback is that we have to manage windows of variable size, i.e., the current-window pointer (CWP) has to be updated to point to the appropriate window when a function returns. This can be solved with two different alternatives: the architecture saves the CWP on a function call to be restored later when the function returns or the programmer (compiler) generates an instruction to update the CWP. Let us discuss them in turn.

To manage multi-size windows implicitly (by the architecture) it is necessary to keep either a copy of the previous CWP when a function is called or to keep the window size to update the CWP. The advantage of keeping the window size is that less bits are required because the number of sizes available is smaller than the number of registers in the register file. The advantage of keeping the pointer is that no ALU operation is required when a function returns to compute the CWP. We select the pointer approach, although the following discussion can

be applied to both of them. Let us see how the pointer can be manipulated when registers are not memory mapped and when they are mapped (see Section 2.3).

If the registers are not memory mapped, then a parallel register file is required to keep track of this information. Since various sizes are available, the number of entries in the parallel register file should be the maximum number of the smallest windows that can be in the register file. The pointer or the size is saved when a function is called and restored when a function returns. If there is overflow, the bottom window is saved together with its pointer in the overflow stack. Saving the window pointer increases the data memory traffic; however, the increase would not be significant because overflow occurs infrequently (see Section 3.8).

If the registers are memory mapped, then it is not possible to store the pointer with the window when an overflow occurs because there is no memory location free between windows. There are two possible solutions: to store the pointer in a third memory stack or to use one of the window registers to hold the pointer. The first solution requires to have a third stack pointer to manipulate the window pointer overflow stack. The second solution causes that the programmer (compiler) cannot use one of the registers in the window because is used by the architecture.

Therefore, although multi-size windows could be managed directly by the architecture, the processor requirements that are needed make this first possibility very unattractive.

The second possibility consists of updating the CWP explicitly by an instruction generated by the compiler. The following two approaches are possible, depending on where the CWP is pointing to:

(1) If the CWP points to the bottom of the current window, then the following operations have to be done:

a) The call instruction specifies the current window size so that the CWP is updated to point to the bottom of the callee window.

b) The number of registers for the largest window is always available in the register file. An overflow exception is generated when the number of free registers is smaller.

c) When the function returns, the CWP is decremented with the smallest window size. Remember that C functions are compiled separately so that the callee has no information about the caller's window size.

d) If the caller uses a larger window, then an instruction is generated after every call (executed after every return) that updates the CWP. This corresponds to the solution (b) given for variable-size windows (see previous section).

Thus, a function always has available the largest size window. Once the compiler know the appropriate window size for the function, then it includes this information in the call instruction to increment the CWP with the exact window size and generates an instruction after every call instruction to decrement it. Observe that this approach is different from the one discussed in the previous section for variable-size windows because the functions that use the smallest size windows do not require any extra instruction.

(2) If the CWP points to the top of the current window, then the following operations have to be done:

a) When a call instruction is executed, the CWP is always incremented with the smallest window size, i.e., the smallest window size is always allocated to the callee.

b) If the callee requires a larger window size, then an instruction is executed to increment the CWP for the appropriate window size.

When the CWP is incremented by (a) or (b) and there are not enough free registers in the register file, then an overflow exception is generated.

c) The return instruction specifies the window size so that the CWP is decremented and the callee window is deallocated. The number of registers for the largest window should always be available in the register file upon return. An underflow exception is generated when the number of registers is smaller.

Although the number of operations or instructions executed for the second approach is the same as for the first, we select the second approach because the generated code size is smaller than the one for the first approach. With the first approach, functions that require a larger window size generates an instruction to decrement the CWP after every instruction call in the function. With the second approach, only one instruction to increment the CWP for each function is necessary.

In conclusion, multi-size windows offer the same advantages as fixed-size windows except for the mapping from virtual register numbers to physical register numbers, and the manipulation of windows of different sizes. However, multi-size windows allow a better utilization of the register file because (1) less data memory traffic is generated during overflows, underflows, and context switching, and (2) less registers are required in the register file to hold a specific number of windows. This is quantified with our measurements described in Section 3.8. Section 3.7

also uses our measurements to deduce the number of different windows available and the number of registers per window. But, first we have to present the advantages of using overlapped windows (Section 3.5) and how the register set is organized with multi-size windows (Section 3.6). There is no architecture known to the author that uses multi-size windows.

3.5 Overlapped versus Non-Overlapped Windows

The main advantage of having *overlapped windows* has been already given in Section 2.8: to reduce the overhead caused by parameter passing. In the following discussion parameter passing refers to the information transferred from the caller to the callee (parameters and return address) and from the callee to the caller (result). If no overlapped registers are available, there are three different alternatives:

(1) To have some registers common to all the functions, i.e., to have some registers which are not part of the window so that they are not saved across function calls. In this case, parameters can be passed through these common registers as has been discussed in Section 2.8. The drawback is the same as when only one window is available and parameters are passed through registers for temporary results: the compiler must decide which parameters are moved to the stack and which ones are moved to registers of the window. Thus, the overhead is not completely eliminated and instruction memory traffic is incremented because of the extra instructions generated.

(2) To have special instructions to pass parameters to the next window. This scheme has three drawbacks:

a) The first drawback is that these instructions have to check if the register file has enough space for the following window. If no space is available the instruction will cause a trap to indicate register file overflow and to make space for the following window. Although only the first instruction is able to generate a trap, every executed instruction of this type has to check if there is an overflow exception. On the other hand, with an overlapped window scheme overflows and underflows are only generated by the call and return instructions.

b) The second drawback is that without overlapped registers every parameter should be transferred to the following window with an explicit move instruction. Looking again at table 2.12, we deduce that 55% of the parameters (constants, global simple variables, and complex parameters) are moved directly to the corresponding overlapped register and that 45% of the parameters need an explicit move instruction. Thus, the number of instructions required for parameter passing is doubled.

c) Finally, the third drawback is that pointers to the next and the previous windows should also be available if we do not want to increment the CWP to map the virtual register number to a physical register number every time that an instruction to transfer an argument is executed.

(3) To pass parameters through the stack. This increases data memory traffic as has been discussed in Section 2.4 and, thus, no advantage is obtained with the multiple windows for parameter passing. Moreover, if we are using the stack pointer as frame pointer (see Section 2.9.2), then all the functions (with at least a statement that performs a function call) need to create a stack frame because of the parameter passing. If parameters are pass through registers and enough registers are available for parameter passing and local simple variables, then only functions with non-local simple variables need to create an activation record in the stack.

On the other hand, overlapped windows have two main disadvantages:

(1) If the register file has N windows and it is organized as a circular-buffer register file (see Section 4.2), then only $N-1$ windows can be used because the registers for outgoing parameters of the last window overlap with the registers for incoming parameters of the first one; therefore, there is always a portion of the register file that is unutilized. This is only true for a circular-buffer register file; Section 4.3 presents an alternative implementation that solves this problem.

(2) The mapping from the virtual register number given by the instruction to the physical register number is easier if the general-purpose register set and the window have the same size because the virtual register number only needs to be concatenated with the current window pointer. This not only requires non-overlapped windows, but also a general-purpose register set without registers that are not part of the window (although it is possible to have some specialized registers as we have discussed in Section 1.6). Thus, the first alternative to solve parameter passing in non-overlapped windows would disappear. It can be shown that the cost of mapping from virtual to physical register numbers when implementing overlapped windows is insignificant (see Section 4.2.4).

In conclusion, overlapped windows offer more advantages for parameter passing than non-overlapped ones. RISC, PYRAMID 90x, and CELERITY C1200 use overlapped registers to pass parameters. C/70 does not have overlapped registers nor common registers to pass parameters, so parameters are passed through the stack.

In the multi-size approach, all window sizes require the same number of overlapped registers. This is because functions are compiled separately so that when a function is compiled there is no information about the caller's window size. Next section discusses how the register set is organized with overlapped

multi-size windows and Section 3.7 presents how many overlapped registers are required.

3.6 The Register Set Organization

This section discusses how the register set is organized. The register set can be divided into four parts: the incoming-parameter registers, the local registers, the outgoing-parameter registers, and some registers common to all the functions, i.e, registers that are not automatically saved across function calls. To pass parameters through registers the outgoing-parameter registers overlap with the incoming-parameter registers of the function to be called as has been proposed in [PATT82] and discussed in the previous section. We assume that the compiler uses the incoming-parameter registers that are free (registers that do not contain any parameter) to store local simple variables, in addition to the local registers. Common registers are called "global" in RISC and PYRAMID terminology. Now, we show that common registers are not necessary.

The register set in RISC consists of the window plus 10 common ("global") registers: six are used for temporary results and four have a special purpose (zero-hardware register, stack pointer, current-window pointer, and first-window pointer). As we have mentioned in Section 2.10, although special registers for global simple variables can be available, global variables cannot be allocated to general-purpose registers. Thus, these "global" registers are used by the Portable C Compiler to store temporary results and environment registers [MIRO82]. Let us discuss this usage.

The temporary results can be stored in the outgoing-parameter registers since these are not used across function calls. This solution has already been implemented by PYRAMID 90x: there are sixteen outgoing-parameter registers that are used as registers for temporary results. However, PYRAMID still has 16 common registers, three of them have a special purpose (program counter, stack pointer, and "a pointer to the base of the current stack frame"*), but no usage has been specified for the other 13 registers. CELERITY C1200 has also 16 common registers, 15 are used as floating-point registers and one as a multiply accumulator.

Thus, the only usage for common registers is to store environment information. Floating-point registers can be referenced with floating-point instructions and, therefore, it is not necessary to have them as part of the general-purpose register set. Since we have already discussed that some environment registers (the program counter, the current-window pointer, and the first-window pointer) should be implemented as specialized registers (see Section 1.6), then we have to

*Little information has been released about PYRAMID so that it is difficult to verify if this pointer is the stack frame. If so, there is no reason of having it as a "global" register because it should be saved on function calls.

decide if they should be mapped in the general-purpose register set or if some special instructions are provided to manipulate them. There are two reasons for not having common registers:

(1) The translation from virtual to physical register numbers is easier if there are no common registers. When common registers are available, it is necessary to detect whether a virtual register number refers to the common register file or to the multiple-window register file and to translate the number to a displacement relative to each file (see Section 4.2.4). If no common registers are available, then virtual register numbers refer only to the multiple-window register file.

(2) Common registers force a larger general-purpose register set because they reduce the number of registers in the window. For instance, for a 16-general-purpose register set, if four general-purpose registers are used for the program counter, the current-window pointer, the first-window pointer, and the stack pointer, then the largest number of registers available to the window is 12 with non-overlapped registers. If we want overlapped registers, this number seems to be too small for all the usages we have discussed in the previous chapter so that a larger register set should be selected (i.e., 32 registers).

Thus, we would prefer to provide some special instructions to manipulate environment registers so that no common general-purpose registers are required. The number of these special instructions is small because the program counter is implicitly updated by other instructions and could be referenced implicitly by a relative addressing mode (although Section 1.6 has shown the advantages of not having this addressing mode); the current-window pointer can be updated implicitly by the call and the return instructions; and the first-window pointer can also be updated implicitly by the instructions to save and to restore a window.

There is one more environment register that it is not required to implement it as specialized register: the stack pointer also used as frame and argument pointer (see Section 2.9). The stack pointer only should be incremented and decremented when an activation record has to be created in the stack because there are not enough registers to keep it. This is expected that occurs infrequently with an adequate choice of the window sizes (see Section 3.7). So, there is no necessity for having the stack pointer as a general-purpose register because it is used for a small number of functions. Thus, we can also implement the stack pointer as a specialized register. Only functions which have part of the activation record in the stack and need the frame pointer to address their local variables, then the return address is saved in the stack (because is not needed during function execution) and the frame pointer is copied in its place. The advantage of loading the stack pointer in a general-purpose register is that it is not necessary to provide a special addressing mode to access local simple variables in the stack because this addressing mode is seldom used.

The RISC zero-hardware register is used to implement the instructions NOOP (*add r0,r0,r0*), clear (*add r0,r0,ri*), move (*add r0,ri,rj*), test (*add r0,ri,r0,{c}*), and compare (*add ri,rj,r0,{c}*); and some addressing modes. If the instruction set implements five more operation codes and the equivalent addressing modes, then no zero-hardware register is necessary.

In conclusion, no common registers are necessary in the general-purpose register set. Outgoing-parameter registers are used to store temporary results and the environment registers (program counter, stack pointer, current-window pointer, and first-window pointer) are implemented as specialized registers. Moreover, some special registers mapped to specific memory locations can be used to allocate global simple variables as discussed in Section 2.10. The general-purpose register set is divided into three parts: the incoming-parameter registers, the local registers, and the outgoing-parameter registers. Note that even without common registers the general-purpose register set size and the window size are not the same because the latter only includes the local registers plus the overlapped registers, i.e., only the registers that must be allocated (deallocated) when a function is called (returns).

The next section determines the number of registers to include in the window. The following factors have to be considered: (1) the number of variables defined by the programmer and by the compiler and their lifetimes (because they can share the same registers), (2) the compiler allocation policy, (3) the data memory traffic overhead introduced by overflows and underflows, (4) the implementation cost of the registers given by the area required and the processor cycle delay, and (5) the number of bits required in the instruction to specify a register address.

Section 3.3 have already discussed the register saving and restoring overhead, Section 3.7 determines the window size based on the number of variables required per function and on the number of bits required for register addressing, and Section 3.8 determines the register file size based on the percentage of overflows. Different register allocation policies have been discussed in Chapter 2 and considered to compute the register saving and restoring overhead. Multi-size windows require the use of an optimizing compiler because to allocate the appropriate window size we need to know how many registers are required by the function before code generation. Also an optimizing compiler is required to use the stack pointer as frame and argument pointer (see Section 2.9). Thus, we cannot use an on-the-fly allocation policy. The cost of implementation is considered only in terms of the number of registers in the register file. Chapter 4 discusses two different implementation alternatives for the register file.

3.7 Window Sizes

This section discusses the number of registers in a window. Our goal is to reduce the data memory traffic caused by local simple variables, parameter passing, and register saving and restoring. If we assume that there are enough windows in the register file so overflow is infrequent, then to reduce the data memory traffic caused by local simple variables we would like to have enough registers in the window to store all local simple variables, i.e., to have to create an activation record only for functions that use non-simple local variables (arrays or structures). However, the number of variables is not the only element to consider to decide the register set size, the decision must balance the implementation cost of the registers and the number of bits required in the instruction to specify a register address (i.e., to have a small window), with the data memory traffic generated by the local simple variables that cannot be allocated to registers (i.e., to have a large window). Consequently, we want to have the smallest window that provides a satisfactory performance.

To determine the window sizes, first we discuss how many registers are required for parameter passing (Subsection 3.7.1); second, we determine the smallest and the largest window sizes depending on the number of functions that need to create an activation record in the stack because there are not enough registers available (Subsection 3.7.2); and third, we study how many window sizes should be available to reduce the number of unused registers in the window so that registers are used efficiently and few unused registers have to be saved and restored during overflow and underflow (Subsection 3.7.3). This will also result in a smaller register file for the same number of windows as discussed in Section 3.8.

3.7.1 Overlapped Registers

This subsection discusses the number of overlapped registers required for parameter passing and result return. The overlapped registers should include one extra register to save the return address to the caller.

A C function can only return one simple variable (a data of a fundamental data type or a pointer). Thus, only one register is required to pass a result back from the callee to the caller. Therefore, at least two overlapped registers should be available: one to store the return address and one to store the return value.

The number of parameters passed from the caller to the callee has been discussed in Section 2.8 and shown in Table 2.13. For the three measured programs, three registers are enough to pass parameters for 98% of the function calls. Since we want to use outgoing-parameter registers to store temporary results, we also need to have enough registers to perform expression evaluation. Section 2.6 has discussed the number of registers required for temporary results and has concluded that two registers are enough for 95% of the expressions.

In conclusion, if the window has four overlapped registers (one to store the return address and three to pass parameters), then 98% of the functions are able to pass parameters through registers. The functions that contain this 2% of function calls with more than 3 parameters, require the creation of a stack frame for the extra parameters. No measurements were made to compute the number of these functions. Let us remark that although the caller needs to create the frame, the callee may have enough registers available for local simple variables and the parameters can be moved from the stack to registers. RISC has six overlapped registers, and PYRAMID 90x and CELERITY C1200 have sixteen, which are also used for temporary results. Our measurements do not justify the increase from six to sixteen registers done by PYRAMID and CELERITY.

3.7.2 Smallest and Largest Window Sizes

To determine the smallest and the largest window sizes in the multi-size window case we consider the number of functions that need to create an activation record in the stack because there are not enough registers available. Table 3.2 shows the percentage of functions that need to create an activation record in the stack for different window sizes due to simple variables, non-simple variables, and their total. Since we have use the Portable C Compiler to collect our measurements, we have assume that one register is used for each local simple variable defined by the programmer, i.e., we have not considered sharing of registers. Thus, it is possible that an optimizing compiler by coloring could reduce the percentage of functions that need to create an activation record in the stack. Observe that the total percentage is not always the sum of the two previous columns because some functions that require a frame for simple variables also require it for non-simple variables. Also, note that the number of activation records created due to non-simple variables is independent of the window size.

We consider the set of window sizes 4, 8, 12, ... because the smallest size should have sufficient registers for the parameters and the return address, and we increase the size by a fixed increment of four.

The smallest window size is 4, i.e., 4 registers for incoming parameters used also to store the return address and the local simple variables, and 4 registers for the outgoing parameters. In this case, the user has available eight general-purpose registers: four for temporary results, three for local simple variables (including parameters), and one for the return address. From Table 3.2 we see that this size is sufficient for 75% of the executed functions.

A possible largest window size in the multi-size window case is 12, i.e., 16 general-purpose registers accessible to the programmer (1 for the return address, 11 for local variables, and 4 for outgoing parameters). From Table 3.2 we see that for this size only 3.3% of the functions need to create an activation record in the stack, 1.9% because of local non-simple variables and 1.4% because of local simple variables.

no. registers lcl+arg	program	% frames		
		non-simple	simple	total
3	NROFF	1.7	10.7	10.8
	SORT	0.0	22.2	22.2
	VPCC	2.2	41.0	41.0
	ALL PROG.	1.9	24.6	24.6
7	NROFF	1.7	0.0	1.7
	SORT	0.0	20.5	20.5
	VPCC	2.2	12.8	15.0
	ALL PROG.	1.9	6.2	8.1
11	NROFF	1.7	0.0	1.7
	SORT	0.0	19.5	19.5
	VPCC	2.2	2.1	4.3
	ALL PROG.	1.9	1.4	3.3
15	NROFF	1.7		1.7
	SORT	0.0	19.5	19.5
	VPCC	2.2	1.7	3.9
	ALL PROG.	1.9	1.2	3.1
19	NROFF	1.7		1.7
	SORT	0.0		0.0
	VPCC	2.2	1.7	3.9
	ALL PROG.	1.9	0.8	2.7
23	NROFF	1.7		1.7
	SORT	0.0		0.0
	VPCC	2.2		2.2
	ALL PROG.	1.9		1.9

Table 3.2. Window Size

This value (16 general-purpose registers) balances the number of bits required in the instruction to specify a register address (4) with the data memory traffic generated by the local simple variables that cannot be allocated to registers (1.4% of the functions require the allocation of some simple variables in the stack). This seems to be an adequate number for a single window, for a fixed-size window, and for the largest multi-size window.

3.7.3 Unused Registers in Window

To determine the set of sizes to include we consider the number of unused registers in the window, as shown in Table 3.3. This number has been computed taking into account only local simple variables defined by the programmer. Two conclusions can be deduced from this Table:

Program	Fixed-size lcl+arg reg. 3	7	11	Multi-size Two-size	Three-size
NROFF	0.8	4.6	8.6	1.5	1.1
SORT	1.6	4.7	7.9	1.7	1.6
VPCC	0.6	3.3	7.0	2.3	1.2
ALL PROGRAMS	0.7	4.0	7.9	1.9	1.1

Table 3.3. Unused Registers in Window

(1) The advantage of having three-size windows (i.e., 4, 8, and 12 registers) with respect to having two-size windows (i.e., 4 and 12 registers) only. Three-size windows use more efficiently the registers available because, on the average, only 1.1 registers are not used as compared to 1.9 for two-size windows. Consequently, on overflow, underflow, and context switching only 1.1 registers (per window) that are saved or restored are not used.

(2) The advantage of having multi-size windows versus having fixed-size windows. If a fixed window of 12 registers were selected, then, on the average, 7.9 registers of each window present in the register file would not be used and would have to be saved or restored on overflow, underflow, and context switching.

	C/70	RISC	PYRAMID CELERITY	two-size	three-size
general-purpose register set	8	32	64	8/16	8/12/16
window size	8	16	32	4/12	4/8/12
overlapped registers	0	6	16	4	4
% of frames (simple)	14	1.2 - 1.4	-	1.4	1.4
unused registers	3	11	27	1.9	1.1
% ftn. pass par. through regs.	-	98.5	100	98.1	98.1

Table 3.4. A Comparison of Existing Processors

To compare our proposal with the existing processors, Table 3.4 summarizes the relevant data from Tables 2.13, 3.2, and 3.3. C/70 has a fixed-window size of 8 to store local simple variables and temporary results. If we assume that two registers are used for temporary results, then six registers are available to store the local simple variables. Our measurements show that six registers are enough for 86% of the executed functions (see Section 2.2.3). The number of unused registers has not been measured, but we can expect that on the average there are about 3 unused registers for each window. This number has been deduced from Table 3.3 observing that for a large window the average number of registers used is three. Neither overlapped nor common registers are available so that parameters are passed through the stack.

RISC has a fixed-window size of 16 to store local simple variables, the stack frame, and the return address. Thus, from Table 3.2 we conclude that between 1.2% and 1.4% of the functions do not have enough registers to store all local simple variables*. The number of unused registers have not been measured, but it is expected to be, on the average, about 11 (deduced as indicated for C/70). Six overlapped registers are available so that 98.5% of the executed function calls pass parameters through registers.

PYRAMID 90x and CELERITY C1200 have a fixed-window size of 32 registers. Our measurements do not show any reason for such a large window because with a window size of 24 registers all local simple variables can be allocated to registers (see Table 3.2) and the number of unused registers to be transferred from/to memory on overflow, underflow, and context switching is expected to be, on the average, about 27 (deduced as indicated for C/70). Sixteen overlapped registers are available so that all the executed function calls pass parameters through registers.

In conclusion, our measurements show that for a register file of three-size windows with four overlapped registers and window sizes of 4, 8, and 12, 75% of the functions have enough with the smallest window, 17% with the medium size window, and 5% with the largest window. Only 1.4% of the functions need to create an activation record in the stack because of local simple variables. On the average, only 1.1 registers of each window are not used and, thus, only 1.1 unused registers have to be saved or restored per window on overflow, underflow, and context switching. Four overlapped registers allow that 98.1% of the function calls pass parameters through registers.

*In reality, the Portable C Compiler does not use the free overlapped registers to store local simple variables so that only 9 registers are used for local simple variables. However, this is a constraint imposed by the compiler, not by the architecture.

Finally, the following considerations have to be made concerning the way the numbers in Tables 3.2 and 3.3 have been computed:

a) We have not accounted for the functions that require a frame because some function they call passes more than three parameters. Our measurements show that 2% of the functions pass more than three parameters; however, no measurements were taken on the number of functions that contain these 2% of function calls.

b) We have not accounted either for the functions that require a frame for the local simple variables that must be allocated to the stack because the address operator is applied to them. These variables cannot be allocated to registers, except if the registers are memory mapped. However, our measurements show that only 0.2% of the data objects are local simple variables with the address operator and that they account for only 0.0013% of the objects executed (see Section 2.3).

c) We have not accounted either for the functions that require a frame because more than four temporary registers are required to perform expression evaluation. We know that two registers are enough to evaluate 95% of the expressions. We do not know how many functions require more than four registers to evaluate expressions and they do not have any free local register in the largest window.

d) The local simple variables included are only those defined by the programmer. Optimizing variables have not been accounted because the Portable C Compiler does not define them. We expect that an optimizing compiler would perform register allocation by coloring (see Section 2.5) so that local simple variables with disjoint lifetimes can share the same registers. Thus, the compiler can use the unused registers in the window for optimizing variables. Observe that, on the average, there are 1.1 unused registers that the compiler can still use.

e) Only functions calls to programmer-defined functions have been measured, i.e., the measurements do not include function calls to library functions.

In spite of this, we believe that the numbers are representative enough to allow us to make the previous comparisons.

3.8 Register-File Size

This section discusses the number of registers of the register file. Three factors are considered to determine the register-file size: (1) the data memory traffic generated by the windows that must be brought to (from) memory because of register file overflows (underflows), (2) the cost of the implementation given by the number of registers required in the register file, and (3) the restrictions imposed by the implementation: the number of registers must be a power of two in a circular-register file (see Section 4.2.1) and a multiple of the general-purpose re-

gister set in a shift-register file (see Section 4.5).

For fixed-size windows, Table 3.5 shows for different register file sizes and three different window sizes —four registers (3 for local variables and 1 for the return address), eight registers (7 for local variables and 1 for the return address), and 12 registers (11 for local variables and 1 for the return address)— the following data: (a) the percentage of overflows, (b) the average number of registers that must be saved per function call, and (c) the average number of windows in the register file. Columns (a) and (b) are used to deduce the data memory traffic generated; column (c) is used to deduce the average number of unused registers in the file and the average number of registers that must be saved on context switching. The percentage of overflows has been computed without taking into account context switching and operating system exceptions, i.e., we have assumed that the register file is dedicated to each of the measured programs. The effects of exceptions and context switching on the register file are discussed in the next section.

From Table 3.5 we conclude that eight is an adequate number of windows to have a low percentage of overflows (2%). The number of registers required in the file depends on the window size. For a window size of 12 (which was shown to be adequate in Section 3.7), the register file has 100 registers (or 128 if it has to be a power of two, in which case, the register file can contain up to 11 windows and the percentage of overflows is approximately 0.4%). For this case we have that, on the average, 4.8 windows are present in the register file and 0.3 registers are saved per function call. Since there are 7.9 unused registers per window (see Table 3.3), the register file will contain on the average 38 unused registers.

RISC has a register file of 128 registers; thus, up to seven windows can be stored in the file. For this case, Table 3.5 shows that the percentage of overflows is 3.2% and that on the average 4.4 windows are present in the register file. As we said in Section 3.2, the percentage of overflows is higher than the one given by Patterson and Séquin [PATT82] due that they measured two benchmark programs. Since there are 11 unused registers per window (see Table 3.4), the register file will contain on the average 48 unused registers, i.e., 38% of the registers in the file are allocated, but unused.

PYRAMID 90x has a register file of 512 registers, that is, up to 15 windows can be stored in the file. Table 3.5 only shows the case for 16 windows; however, the numbers should be very similar: a percentage of overflows of about 0.1% and an average of 6.7 windows present in the register file. Since there are 27 unused registers per window (see Table 3.4), the register file will contain on the average 181 unused registers, i.e., 35% of the registers in the file are allocated, but unused.

number of windows in RF	program	% ovf	avg. reg. saved/call lcl+arg reg. 3	7	11	average windows in RF
2	NROFF	51.1	2.0	4.1	6.1	2.0
	SORT	19.8	0.8	1.6	2.4	2.0
	VPCC	41.8	1.7	3.3	5.0	2.0
	ALL PROG.	46.2	1.8	3.7	5.5	2.0
4	NROFF	18.2	0.7	1.5	2.2	3.2
	SORT	0.1	0.0	0.0	0.0	3.1
	VPCC	12.1	0.5	1.0	1.4	3.1
	ALL PROG.	15.0	0.6	1.2	1.8	3.1
5	NROFF	9.0	0.4	0.7	1.1	3.6
	SORT	0.0	0.0	0.0	0.0	3.4
	VPCC	7.9	0.3	0.6	0.9	3.5
	ALL PROG.	8.3	0.3	0.7	1.0	3.6
7	NROFF	3.1	0.1	0.2	0.4	4.5
	SORT	0.0	0.0	0.0	0.0	3.8
	VPCC	3.5	0.1	0.3	0.4	4.3
	ALL PROG.	3.2	0.1	0.3	0.4	4.4
8	NROFF	1.9	0.1	0.2	0.2	4.8
	SORT	0.0	0.0	0.0	0.0	4.1
	VPCC	2.5	0.1	0.2	0.3	4.8
	ALL PROG.	2.2	0.1	0.2	0.3	4.8
12	NROFF	0.1	0.0	0.0	0.0	5.5
	SORT					5.7
	VPCC	0.7	0.0	0.1	0.1	6.6
	ALL PROG.	0.4	0.0	0.0	0.0	6.0
16	NROFF	0.0	0.0	0.0	0.0	5.6
	SORT					5.7
	VPCC	0.1	0.0	0.0	0.0	8.1
	ALL PROG.	0.1	0.0	0.0	0.0	6.7

Table 3.5. Fixed-Sized Windows

CELERITY C1200 has eight register files. Each one is associated to an active process so that to switch context from one process to another (among these eight) only one pointer has to be changed. This organization is discussed in Section 3.9.2 when multiple private register files are considered. Here, we only consider the register file that is available to one process. In this case, the register file is identical to the one described for PYRAMID 90x.

C/70 has a register file of 1024 registers, i.e., it can contain 128 windows. As we said in the Section 3.1, the large number of windows is a consequence that C/70 does not want to handle overflow exceptions. Since a large register set occupies a large area in a VLSI implementation (33% of the chip in RISC [KATE83])

or requires many chips in the MSI/LSI case, register sets of 512 or 1024 registers are only justified if overflows are extremely slow.

number of registers in RF	program	two-size (3/11)			three-size (3/7/11)		
		% ovf	avg. reg. saved/call	avg. reg. in RF	% ovf	avg. reg. saved/call	avg. reg. in RF
32	NROFF	8.5	0.9	20.5	4.8	0.3	19.9
	SORT	0.3	0.0	27.1	0.3	0.0	27.0
	VPCC	12.7	1.5	21.4	9.7	1.0	20.8
	ALL PROG.	10.2	1.1	21.0	6.9	0.6	20.5
48	NROFF	2.2	0.2	29.4	1.1	0.1	26.6
	SORT	0.1	0.0	30.8	0.1	0.0	30.5
	VPCC	4.3	0.5	30.5	2.9	0.3	29.2
	ALL PROG.	3.1	0.3	29.9	1.9	0.2	27.9
64	NROFF	0.6	0.1	34.9	0.1	0.0	30.2
	SORT	0.0	0.0	34.4	0.0	0.0	34.2
	VPCC	2.1	0.2	38.2	1.3	0.1	37.0
	ALL PROG.	1.3	0.2	36.4	0.6	0.1	33.3
80	NROFF	0.1	0.0	38.4	0.0	0.0	30.8
	SORT	0.0	0.0	39.4	0.0	0.0	38.7
	VPCC	1.1	0.1	44.2	0.6	0.1	43.3
	ALL PROG.	0.5	0.1	41.0	0.3	0.0	36.6
96	NROFF	0.0	0.0	39.3			30.9
	SORT	0.0	0.0	42.4	0.0	0.0	41.8
	VPCC	0.6	0.1	48.4	0.3	0.0	48.8
	ALL PROG.	0.3	0.0	43.5	0.2	0.0	39.2

Table 3.6. Multi-Size Windows

Table 3.6 shows the equivalent information to Table 3.5 for two-size and three-size windows. For the three-size case the best candidate for the register-file size is 64 because it generates less than 1% of overflows (0.6%), it is a multiple of 16 which is convenient for the shift-register-file implementation, and is a power of two which is convenient for the circular-buffer-register-file implementation. In this case, the average number of registers saved per function call is 0.1 and the average number of registers present in the register file is 33.3. Consequently, the three-size implementation is significantly better than the fixed size: the number of registers is reduced to half, the percentage of overflows and the number of registers saved per overflow to one third, and the average number of registers being used from 52 to 33. This last figure is important because it corresponds to the number of registers that have to be saved in context switches. Therefore, one of the drawbacks of using a register file [HENN84], (that the process switching time is increased "by dramatically increasing the processor state") is reduced.

From another perspective we can see the advantages of three-size windows if we consider a register file of 64 registers (in this case the register file can contain up to 5 windows of 12 registers). With respect to a register file with 5 fixed-size windows of 12 registers, multi-size windows reduce the overflows from 8.3% to 0.6%, the average number of registers saved per call from 3.6 to 0.1, and the number of registers used from 44 to 33.3.

We also observe the advantage of having three-size windows with respect to the two-size window case in all three aspects: percentage of overflows, average number of registers saved per call, and average number of registers present in the register file. For instance, for 64 registers the percentage of overflows is reduced form 1.3% to 0.6%, the average number of registers saved per call from 0.2 to 0.1, and the average number of registers present in the register file from 36.4 to 33.3.

In conclusion, our measurements show that for three-size overlapped windows and for a 64 register file, overflows occur in less than 1% of the calls, only 0.1 registers have to be saved per function call, and on the average 33.3 registers are present in the register file. Three-size windows allow a better utilization of the register file than fixed-size windows because a smaller register file can be used and, therefore, the implementation cost and the process state to be saved on context switching are both reduced.

3.9 Exceptions and Context Switching

One of the drawbacks given by Hennessy [HENN84] for using a register file with multiple windows is that the process switching time is increased "by dramatically increasing the processor state." Since more registers are available in the processor for multiple-window architectures than for single-window architectures, the amount of processor state to be saved is larger. For this reason, in this section we discuss the effects of exceptions and context switching on the register file.

When the processor execution cycle is broken because an exception has occurred, then the current processor state has to be saved and a new one has to be loaded to service the exception. The processor state includes the register set (general-purpose registers plus specialized registers), privileged registers (such as the processor status word), and mapping information to translate virtual memory addresses to physical memory addresses. In this section we only discuss the effects of exceptions and context switching on the register file. Memory management support is not discussed in this thesis.

Although a context switch occurs as a result of an exception, here we use the term *exception* to refer to the event in which the processor control goes to an exception handler (or to a kernel process for concurrent operating systems) and returns to the original process, and the term *context switching* to refer to the event in which a new user process is scheduled.

On exceptions the processor state can be saved (loaded) partially or totally. If it is saved partially, then the exception handlers must not destroy any part of the previous process state information (this can be guaranteed by the architecture or by "trusted" handlers). If it is saved totally, then the processor state associated to the running process is better protected.

On context switching we have a similar problem. The context of the previous process can be left in the processor or can be completely removed. If it is left in the processor, then the architecture should protect it against possible errors generated by the current process and it should be possible to identify the information available per process.

If the processor state is totally saved, then two drawbacks have to be considered. First, the overhead caused by saving the whole processor state on each exception or context switch can be a bottleneck in the system depending on the amount of information to be saved and the frequency of exceptions and context switches.

Second, due to the frequent change of processor state, the register file might not be used effectively. If the average number of function calls between overflows is larger than the average number of function calls between exceptions or context switches, then it is probable that the register file can be made smaller because it never has the opportunity of becoming full.

Thus, to determine the degradation on the usage of the register file due to exceptions and context switches we have to study (1) the overhead generated on exceptions and context switches due to the register file saving, and (2) the average number of function calls between exceptions and context switches. This is done in Subsection 3.9.1.

Once we have determined the feasibility of saving totally or partially the contents of the register file on every exception and context switching, then we have to determine how the register file has to be organized. Two possible organizations are studied:

(1) Private Register File. In this case, the whole register file is devoted either to one user process or to one exception handler at a time. If only one register file is available, then it must saved on every exception and context switch. To avoid this, the processor can have more than one register file. Section 3.9.2 discusses the private-register-file organization.

(2) Shared Register File. In this case, the register file is shared either (i) among the user process and the exception handlers so that the register file has only to be saved when a new user process is scheduled, or (ii) among several user processes and the exception handlers so that the previous contents does not have

to be saved even on context switching. Section 3.9.3 discusses the shared-register-file organization.

Finally, one remark has to be made about where the register file contents is saved. If we adopt a similar solution to the one implemented for single-window architectures (i.e., the registers are saved in the process control block of the current process), then when the processor state has to be restored, there are two possible alternatives:

(a) In the first one, the whole register file contents is restored. This is a bad policy because if the program is increasing its stack depth (performing more function calls than returns), some windows are transferred again to memory without having been used.

(b) In the second one, only the top window is restored. Since we do not know the current window size when the processor state is restored, we have to load the largest window in the register file. In this case, the complexity of the overflow and of the underflow handlers is increased. When an overflow is detected, the overflow handler has to save the window in the overflow stack indicating that some previous windows are still in the process control block. When an underflow exception is detected, the underflow handler has to check if the window has to be restored from the process control block or from the window overflow stack.

Therefore, to be able to load only the top (largest) window and to continue program execution transparently to the context switch event, the processor state has to be saved in the window overflow stack.

In this case, the page (or pages) corresponding to the window overflow stack must be present in main memory. This is because if the processor state has to be saved before handling a context switch and the overflow stack is not in main memory, then the processor will be idle during the time required to service the page fault handler because two user processes cannot share the register file while a context switch is performed. This is also true for an exception. Thus, the number of pages allocated per process is increased. Note that for the single-window case, registers are saved to the process control block which is stored permanently in main memory.

3.9.1 Overhead and Frequency

This subsection discusses the overhead generated on exceptions and context switches due to the register file saving, and the average number of function calls between exceptions and between context switches to determine the effects of multiprogramming on the performance of the register file. We show (1) that the register file should not be saved totally when an exception occurs because of the high frequency of exceptions and the low average service time, and (2) that the re-

gister file is fully utilized because the average number of function calls between context switches is greater than the average number of function calls between overflows. Since no measurements can be made using our register set, the effects of multiprogramming on the register file are determined with measurements made on two other machines. One was made by this author on PDP-11/60 [HUGU81] and the second was made by Emer and Clark on VAX-11/780 [EMER84].

exception type	frequency			service
	average in 2' int.	total no.	%	time avg. (μs)
DZ-11 (1) multiplexor (display)	8996.47	512799	28	387.62
DZ-11 (2) multiplexor (display)	5277.75	300832	16	391.41
Clock	6000.07	342004	19	544.47
Disk	1270.46	72416	4	1086.95
DZ-11 (1) multiplexor (keyboard)	576.23	32845	2	1118.67
Printer	192.56	10976	1	2014.80
Others (floppy disk, console, ...)	1032.82	58871	3	
Total interrupts	23346.36	1330743	73	479.91
EMT instruction	8524.21	485880	27	
Others (TRAP, priv. instr., ...)	129.02	7354	-	
Total traps	8653.23	493234	27	
Total exceptions	31999.59	1823977		

Table 3.7. Measured Exceptions on PDP-11/60

A software monitor was implemented by this author on the RSX-11M operating system running on the PDP-11/60 of the Computer Science School at the *Universitat Politècnica de Barcelona*. Statistics about the real system load during two hours were collected. Table 3.7 reproduces the number of exceptions that the system handles, on the average, during two minute intervals. Observe that there is a clock interrupt every 20 ms. because in Europe the power line frequency is 50 Hz. Exceptions are divided into two classes: traps (synchronous) and interrupts (asynchronous). During these two hours the processor was 21% of the time in system mode, 23% in user mode, and 56% idle*. From the table we deduce that, on the average, the interval between two exceptions is 3.95 ms.

This average interval between two exceptions (3.95 ms) does not reflect only the time that processor is executing user code, but also includes the processor idle time. To have a rough estimation of the interval time between exceptions, let us consider only the time that the processor is busy, i.e., we assume that the processor has switched from busy to idle only once during this 2-hour inter-

*The idle time is a consequence of the type of load that the PDP-11/60 had back in 1981: mostly first-and-second-year students learning PASCAL and MACRO-11 (PDP-11 assembly language), i.e., programs with very low execution time.

val. We also assume that all exceptions have occurred when the processor was in user mode, but we only account clock interrupts for this interval. In this case, the average interval between two exceptions is 0.94 ms.

Now, we want to determine the average service time for exceptions, interrupts, and traps. The average service time for an exception is deduced dividing the time that the processor has been in kernel mode by the number of exceptions given in the Table. Thus, on the average, the exception service time is 829 μs. The average service time for an interrupt handler was measured and is given in the Table: 480 μs. If we multiply the interrupt average service time by the total number of interrupts, we can deduce that interrupts account for 73% of the exceptions and 42% of the time that the processor has been in system mode, and that traps account for 27% of the exceptions and 58% of the time. The 58% has beed computed from the difference of the total exception service time minus total interrupt service time. From here we can also deduce the average service time for a trap (mainly system calls): 1,771 μs.

Given the time between exceptions and the average service time, we would like to know how many call instructions have been executed during this interval and in the exception handlers. The average instruction execution time for PDP-11/60 instructions is given in [SNOW81]: 1.578 μs. Thus, on the average, an interrupt handler executes 304 instructions, a trap handler executes 1,122 instructions, and 596 instructions are executed between two exceptions. Shustek [SHUS78] has measured statically 10,000 lines of code from the RSX-11M operating system. He shows that 6.3% of the instructions are function calls (*jsr*). Since no dynamic measurements are know to the author, we use this percentage to conclude roughly that, on the average, 20 call instructions are executed for an interrupt handler, 73 for a trap handler, and 39 in the interval between two exceptions when no idle time is considered.

To determine the overhead generated by saving the register file on exceptions, we consider a 64-register file with three-size windows. In this case, on the average 33.3 registers have to be saved (see Table 3.6). If we assume that, on the average, 41 instructions are required to perform the register file saving (33) and the loading of the smallest window for the new process (8)*, then the overhead introduced on every exception is about 7%. The overhead has been computed considering that 41 instructions have to be executed for every 596 instructions executed between two exceptions.

*Usually a new processor state has to be loaded for the exception handler. In this case, we only consider the smallest register window. This can be used by the architecture to pass some parameters to the handler. For instance, for the program error handler, the architecture can initialize an incoming-parameter register with the type of error that has been detected.

Moreover, in this case the register file cannot be used very efficiently because the average number of function calls executed between two exceptions (39) is lower than the average number of function calls between exceptions (167 for a 64-register file with three-size windows; see Table 3.6). Also, the number of function calls executed for the interrupt handlers (that account for 73% of the exceptions), let us conclude that the register file is not very disturbed when it is shared among the user process and the exception handlers (see Section 3.9.3) because, on the average, 20 call instructions are executed for an interrupt handler.

In this study no measurements were made about context switching frequency.

Emer and Clark have monitored the performance of a VAX-11/780 under a specific timesharing workload. They measured the number of instructions executed between exceptions and between context switches: 637 instructions are executed between hardware and software interrupts, 2539 instructions are executed between software interrupt requests, and 6418 instructions are executed between context switches. They also measured the instruction frequency by group, 4.5% of the executed instructions belongs to the group of subroutine call and return, and 2.4% to the group of procedure call and return. If we assume that there are the same number of calls than returns, then 3.45% of the executed instructions are calls. Thus, on the average 220 call instructions are executed between context switches, but only 22 between exceptions.

Therefore, the overhead introduced on every context switch is not significant: 0.6% (computed similarly to the overhead for exceptions), and the register file is well utilized between context switches because the average number of function calls (220) is higher than the average number between overflows (167). Moreover, for exceptions we obtain similar results to the ones detected in the previous study: 6.4% of overhead and a lower average number of function calls (22) executed between exceptions.

In conclusion, the register file should not be saved totally every time that an exception occurs because of the high frequency of exceptions (mainly interrupts) and the low average interrupt service time. Only the specialized registers (current-window pointer, first-window pointer, program counter, and stack pointer) should be saved, although this can also be avoided if a different physical register exists for each processor state. Moreover, the register file is utilized efficiently if it is saved only during context switching because in this case the average number of call instructions executed between context switches is higher than the average number of call instructions executed between overflows.

3.9.2 Private Register File

This subsection discusses three alternative organization approaches for the register file when the whole register file is devoted either to one user process or to one exception handler at a time. The three approaches are: to have only one register file; to have two register files, one for the user processes and a second for the exception handlers (operating system functions); and to have multiple register files so that no register file saving and restoring is involved on context switching if there are less active processes than register files.

If only one register file is available, then its contents must be totally saved every time that there is an exception. As we have discussed in the previous subsection, the exception rate (mainly interrupts) makes this approach unattractive. Moreover, to service an exception we have to wait for saving the whole register file first; this time can be too large for some applications. The advantages of this approach is that it is simpler to implement because only the area for one register file is required and that it is easier to manipulate because only one process uses the register file at a given time.

A second alternative is to have two register files, one for the user processes and a second for the exception handlers. In this case, the user register file has to be saved only on context switching. The advantages of this approach are that (a) the user register file is not disturbed with the exception events; and that (b) when an exception occurs, if a user process is active, then the exception handler can use the kernel register file without having to perform any user register file saving; however, if an exception handler is already active, then the kernel register file should be saved first. Thus, there is no delay to process an exception when the processor is executing user code because no register saving has to be performed. The processor is executing user code more than kernel code so that when an exception occurs, the probability of using the kernel register file is lower than the probability of using the user register file. However, in this case, each exception handler requires its own window overflow stack. Since the exception handlers are dispatched in LIFO order, the kernel register file can be shared among them so that only one common window overflow stack is required for the operating system.

The drawbacks of having two register files are that (a) it does not optimize the usage of the register file because exception handlers use only part of it and user processes use the other part, and that (b) it increases the processor area dedicated to register storage because two register files are required. However, since the register file size has been reduced to 64 registers, it is still possible to have a VLSI implementation (Katevenis [KATE83] has shown that 128 registers can be implemented using 33% of the chip area).

Finally, if multiple register files are available, then no register file saving and restoring is involved on context switching if there are less active processes than register files. In this case, context switching consists only of changing a pointer to the current register file. The advantage of this approach is the fast context switching time. The drawback is that it is impossible to implement in VLSI because of the large area required and it would require many chips in a MSI/LSI implementation. This cost can only be justified if the context switching time is very critical for a few number of applications (e.g., real-time applications).

TMS 9900 [OSBO81] is an example of a machine with multiple single windows. A new ***register bank*** is allocated to each active process in the system. However, the ***register banks*** are not processor registers, but memory locations. This approach provides a fast context switching mechanism because no register saving and restoring is involved. However, a memory reference is generated for each ***register*** access, although the program can use short addresses to refer these ***registers***.

CELERITY C1200 has available eight private register files of 512 registers each, i.e., 4K of registers are provided. Context switching is performed by changing the process identification number that defines which process is active and, therefore, which register file has to be used. No register saving and restoring is involved if less than eight processes are active.

In conclusion, if the register file is private to each process, then two register files should be available in a VLSI implementation, one for the user processes, only saved (and loaded) on context switching, and a second for the exception handlers. To have only one register file private to each process is not recommended because of the high frequency of exceptions (mainly interrupts).

3.9.3 Shared Register File

This subsection discusses two alternative approaches for the register file when it is shared either (1) among several user processes and the exception handlers so that the previous contents does not have to be saved on context switching, or (2) among the current user process and the exception handlers so that the register file contents have to saved only when a new user process is scheduled. Let us discuss them in turn.

Register File Shared Among Several Processes

To share the register file among all active processes, it must be possible to identify the process to which each window belongs to, because windows must be saved in each process address space. Since processes are not scheduled in LIFO order, the operating system cannot have a common overflowing area for all the processes. The operating system cannot have either an overflow stack for each process in its address space because it would increase significantly the operating

system memory space.

Likewise, windows cannot be allocated in LIFO order in the register file. For instance, let us assume the following situation shown in Figure 3.1: process A was running and had allocated some windows in LIFO order in the register file; process A expires its quantum so that the processor is switched to process B; process B keeps executing and allocating windows in LIFO order from the register file until its quantum finishes. Now, the processor is given back to A, but A cannot allocate any more windows in LIFO order because they have been allocated to B. Thus, windows in the register file should be allocated randomly so that the register file is implemented as a register cache.

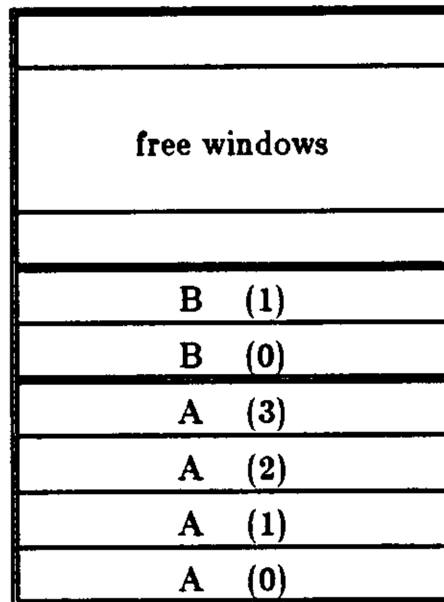


Figure 3.1. Shared Register File with LIFO order

The most attractive mapping scheme for the register cache is set associative. With a direct mapping, different processes could be sharing the same registers, even though some registers are free in the cache. The fully associative mapping is too expensive to implement if a large number of registers are available in the cache.

To avoid to perform a memory access every time that a register is written a write-back policy has to be adopted. When a function returns, registers should be marked as unused to be able to find free registers in the cache and to avoid to have to save them in memory. Registers should not be memory mapped because it would be too expensive to check if the register associated to a memory location is present in the cache.

Since the cache replacement algorithm does necessarily save the registers in LIFO order, then each register or window need to have associated a memory location indicating where they must saved. Therefore, the cache mapping unit should have associated not only a tag register with a unique process identification number (PIN), but also a memory address to be used by the replacement algorithm. The memory address should be the physical memory address because when the register needs to be saved, the memory mapping for the process to which the register belongs to, may be unavailable. In this case, all pages associated to the window overflow stack must be present in main memory rather than only the pages associated to the top windows. This is a consequence of the write-back replacement policy.

The cache mapping unit for the register file cache can be the window or the register. If the mapping unit is the register, then the mapping function can use the virtual register number or the physical register number. Let us discuss them in turn.

(1) The mapping unit is the window. In this case, the windows should be all of the same size and non-overlapped. The windows should have the same size because each window needs to have associated the above mentioned tags (PIN and memory address). The windows should be non-overlapped because they are allocated randomly in the cache register, not sequentially. Thus, two architectural features (multi-size and overlapped windows) which have been shown to be advantageous to support efficiently function calls cannot be implemented with the register cache if the mapping unit is the window. Moreover, the average number of unused registers is larger because of the use of fixed-size windows. These drawbacks are solved when the mapping unit is the register.

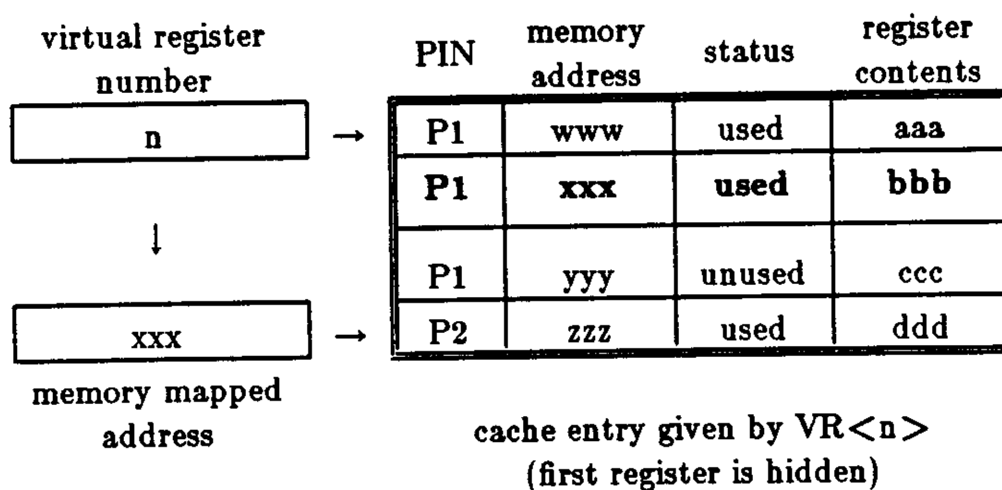


Figure 3.2. Virtual Register Numbers as Mapping Function

(2) The mapping unit is the register and the mapping function is the virtual register number. Since registers for different windows have the same virtual register number, when a call instruction is executed, registers associated to the old window should be hidden until the function returns. This can be implemented using the memory address associated to each register and matching not only the PIN, but also this address. This is implies to compute the memory address for each register referenced. For instance, Figure 3.2 shows a 4-way associative cache entry for a given virtual register number (VR<n>). Two processes, P1 and P2, have registers in the cache. P1 has two registers, one for the current window and a second for a previous window so that this is hidden. P2 has only one register associated to its current window. Observe that P1 had another register in the cache that is marked ***unused*** because the function has returned and, thus, the registers belonging to the window have been destroyed.

Since the compiler tends to use the same register numbers, some cache entries are more required than others. To avoid this, a skewed mapping should be defined so that the same register number is mapped to different cache locations.

(3) The mapping unit is the register and the mapping function is the physical register number. Since in a circular buffer the physical register numbers are calculated modulo the register file size, we need to make hidden registers associated with numbers that wrap around. It is not clear how such a mechanism can be implemented.

In conclusion, to be able to use multi-size overlapped windows the mapping unit should be the virtual register number, but the memory address associated to each register and the tags with the PIN increase significantly the amount of storage required. To decrease the amount of storage required, the window could be used as the mapping unit; however, this solution is discarded because fixed-size windows with non-overlapped registers should be used. Therefore, a register cache is not a good solution to share the register file among processes because it increases the complexity of the processor and the area required for register storage.

Register File Shared Among Running User Process and Exception Handlers

The register file can be shared among the running user process and the exception handlers so that it contents is only saved on context switching. When an exception occurs a non-overlapping window must be allocated so that the outgoing-parameter registers are not destroyed. However, for system call exceptions windows can also be overlapped to allow parameter passing from the user to the system. The current-window pointer is saved automatically together with other process environment information (program counter, processor status word).

The kernel uses the same first-window pointer to detect overflows.

The advantage of this approach with respect to the duplication (user/kernel) of the register file given in the previous subsection is that the register file is used more efficiently because the whole register file is available independently of the processor mode. The drawback is that if the user is using the last window available in the register file, then at least one window should be saved before being able to handle the exception. Table 3.8 shows the percentage of functions that have allocated *n* registers in the file when neither exceptions nor context switches are considered using three-size overlapped windows and a file of 64 registers. As we can see, the smallest size window (8 registers) cannot be allocated for 5% of the functions.

If we want to have always at least one window available to handle interrupts, then we can change the overflow detection condition (see Section 4.2.2) so that an overflow for user processes is generated not when there are no more free registers, but when there is only one window left (probably the smallest window size plus the number of overlapped registers should be enough). The drawback in this case is that we are not using the whole register file for the user process. The advantage is that we do not have to wait for saving the largest window when an exception occurs and there are no more free registers This is only possible with a circular-buffer register file (see Section 4.2.3), not with a shift-register file (see Section 4.5).

number of registers	NROFF %	SORT %	VPCC %	ALL PROGRAMS %	accum.
8			0.0	0.0	0.0
12	0.1		0.7	0.4	0.4
14	4.3	0.0	0.9	2.7	3.1
20	7.6	0.0	3.0	5.3	8.4
24	9.5	0.0	4.2	6.9	15.3
28	17.0	10.9	10.3	13.9	29.2
32	15.0	42.3	10.7	13.6	42.8
36	13.5	2.2	12.9	13.0	55.8
40	12.6	8.3	11.9	12.2	68.0
44	6.7	22.4	10.1	8.6	76.6
48	5.3	1.2	11.1	7.8	84.4
52	3.0	6.1	8.9	5.7	90.1
56	3.3	4.2	7.2	5.1	95.2
60	1.9	0.5	5.4	3.4	98.6
64	0.4	1.9	2.9	1.5	100.1

Table 3.8. Register File Depth

In conclusion, the register file can be shared among the running user process and the exception handlers.

We do not have information about how C/70 and RISC handle exceptions, but both might share the register file among the running user process and the exception handlers. Although people from PYRAMID 90x claim that they have an excellent and original mechanism to support efficiently context switching, this author has not been able to obtain any information about it.

3.10 Conclusions

In this chapter we have confirmed that multiple windows offer an excellent support for function calls independently of the instruction set design (reduced or complex). When multiple windows are available, the data memory traffic is reduced by a factor of 2.7 or 3.5 with respect to the case in which a single window is available.

The advantages and disadvantages of fixed-size and variable-size windows have been studied and an intermediate solution have been proposed: multi-size windows. Multi-size windows combine the advantages of both approaches: small register addresses are required in the instruction, the bus size can be made independent of the register file size, no extra instruction is required to execute a function call if the smallest window is needed, less data memory traffic is generated during overflows, underflows, and context switching, and less registers are required in the register file to hold a specific number of windows.

Using our measurements we have concluded that three-size overlapped windows (with four overlapped registers and window sizes of 4, 8, and 12) and a register file of 64 registers balances some of the mentioned factors to determine the register set design:

(1) The number of local simple variables required: 75% of the functions have enough with the smallest window, 17% with the medium size window, and 5% with the largest window. Only 3% of the functions need to create an activation record in the stack, 1.9% because of local non-simple variables and 1.4% because of local simple variables.

(2) The data memory traffic overhead: overflows occur in less than 1% of the calls and, on the average, only 0.1 registers have to be saved per function call.

(3) The number of bits required to address a general-purpose register: only 4.

(4) The implementation cost given by the storage area required: only 64 registers.

Furthermore, on the average, only 1.1 unused registers have to be saved or restored per window on overflow, underflow, and context switching, and the number of registers to be saved during context switching is not very large: on the average, only 33.3 registers are present in the register file.

Finally, we have considered the effects of multiprogramming on the register file and shown that (1) the register file should not be saved totally when an exception occurs because of the high frequency of exceptions and the low average service time, and (2) the register file is fully utilized because the average number of function calls between context switches is greater than the average number of function calls between overflows. Consequently, two different organizations have been studied: private register file and shared register file. We have concluded that with a private organization, two register files are required one for the user processes and the other for the exception handlers, and that with a shared organization, the register file is shared only among the user and the exception handlers, not among several user processes.

CHAPTER 4
REGISTER-FILE IMPLEMENTATION

4.1 Introduction

This chapter studies the implementation of a register file. The implementation is discussed at the level of the algorithms required to manipulate it. As we said in Section 2.9, there are two pointers to manage the register file: the current-window pointer (CWP) and the first-window pointer (FWP). We study how these pointers are modified on function call and return, on register file overflow and underflow, and during context switching. We also discuss how translation is performed from virtual register to physical register numbers.

Section 4.2 discusses the implementation of multi-size windows using a *circular-buffer register file*. Katevenis [KATE83] has already shown how a circular buffer is used for fixed-size windows and memory mapped registers. However, he affirms that both the register file size and the window size must be a power of two to avoid integer divisions by arbitrary constants. We show that this is only true for the register file size, but not for the window size. We extend the implementation to the multi-size case.

One of the drawbacks for using a register file approach given by Hennessy [HENN84] and admitted by Patterson [PATT85] is that the register file size influences the basic processor clock cycle. The larger the register file is (and, therefore, the more windows are available), the slower the clock cycle is. This is a consequence of the implementation of the register file as a random memory, i.e., access to any register location is provided, not only to the current window. Section 4.3 presents an alternative implementation to the circular-buffer organization called the *shift-register file* that makes the clock cycle independent of the register file size. It is also shown how the shift-register file is organized to implement fixed-size windows with overlapped registers (Section 4.4) and multi-size windows (Section 4.5).

4.2 Circular-Buffer Register File

This section presents how multi-size windows are managed in a register file with a circular-buffer organization. A circular-buffer implementation avoids the relocation of the windows in the register file when an overflow occurs (see Section 3.1). This section is organized as follows: first, we study what operations are performed on the CWP to allocate and deallocate windows; second, we present how

overflow and underflow are detected and what operations are performed on the FWP; third, we discuss how context switching is performed; fourth, we show how translation from the virtual register number to the physical register number is done and we compare it with the translation performed for fixed-size windows in C/70, RISC, PYRAMID 90x, and CELERITY C1200; and finally, we illustrate how memory mapped registers are implemented.

When an overflow occurs, the oldest (first) window in the register file is saved in the window overflow stack. Let us assume that the window overflow stack starts at the top (higher addresses) of the memory space, that the stack grows downwards, and that the memory address and the registers have 32 bits. The following notation is used:

N	Number of registers in the general-purpose register set
R	Number of registers in the register file
$M_0, M_1, ..., M_s$	Number of registers per window (M_0 is the smallest window and M_s is the largest one).
P	Number of overlapped registers
C	Number of common registers

n and *r* are the number of bits required to address the general-purpose register set and the register file.

Independently of whether the registers are memory mapped, the following registers are required to manage the register file (see Figure 4.1):

CWP	Current-Window Pointer (32-bit memory address)
CWRF	Current Window in Register File (r-bit register address; if there is a single window size and it is a power of two, then the last bits will be zero)
FWP	First-Window Pointer (32-bit memory address)
FWRF	First Window in Register File (r-bit register address)

The registers CWP and FWP can also be used as CWRF and FWRF if the number of registers in the register file is a power of two. Let us use both names to make explicitly when we are referring to the pointers in the memory stack and when we are referring to the pointers to the register file, even though below (in Subsections 4.2.1 and 4.2.2) it is shown that they might be the same.

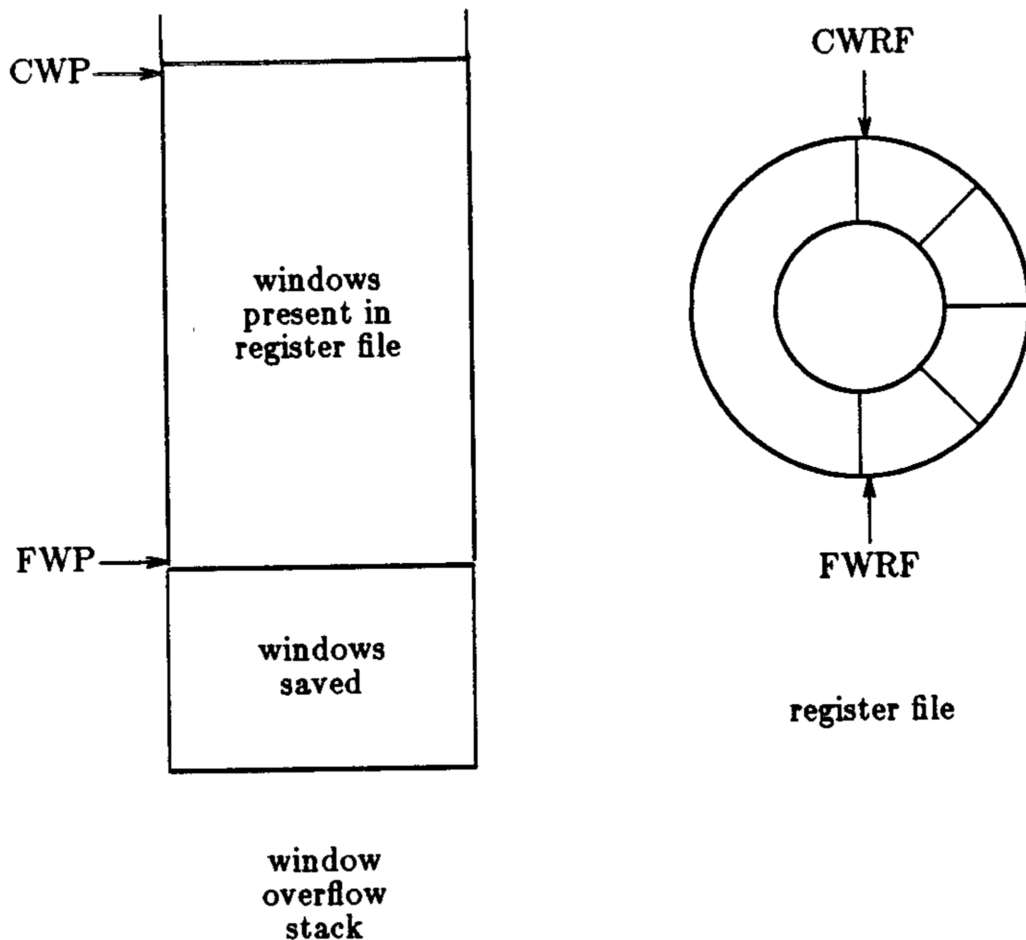


Figure 4.1. Circular-Buffer Register File

If registers are memory mapped, the CWP is used to compute the memory address associated to a given virtual register number (see Section 2.3). Since the CWP is a specialized register (see Section 3.6), an instruction has to be provided to perform this operation. If registers are unmapped, then although an r-bit register (CWRF) would be enough to translate virtual to physical register numbers, the CWP is required to detect overflows as we will discuss in Subsection 4.2.2.

Since the registers have 32 bits and the stack is word aligned, then the last two bits of CWP and FWP are zero. Thus, the operations discussed below on these pointers refer only to its <31:2> bits. However, 30-bit operations are not necessarily performed on both pointers because the window overflow stack has allocated only the last pages of the memory address space and, therefore, the higher order bits of both registers are always one.

4.2.1 Window Allocation and Deallocation

As we have discussed in Section 3.4, the smallest window is automatically allocated when a new function is called. If the callee requires more registers, then the window is expanded by an instruction generated by the compiler. The window is deallocated when the function returns; the return instruction contains the window size. Thus, the following operations are performed:

(1) on function call:

$CWP \leftarrow CWP - M_0$
$CWRF \leftarrow (CWRF + M_0) \textbf{ mod } R$

(2) on expanding the window size up to M_i:

$CWP \leftarrow CWP - (M_i - M_0)$
$CWRF \leftarrow [CWRF + (M_i - M_0)] \textbf{ mod } R$

Note that the operation $(M_i - M_0)$ does not have to be performed because the difference is known in advance. In the following subsections, we use Z_i to refer to this difference: $Z_0 = M_0$, and $Z_i = (M_i - M_0)$, $i = 1, ..., s$.

(3) on function return:

$CWP \leftarrow CWP + M_i$
$CWRF \leftarrow (CWRF - M_i) \textbf{ mod } R$

If R is a power of two and the initial values of CWP and CWRF are (1,1,...,1,0,0) and (0,0,...,0)*, then the register CWRF corresponds to the one's complement of the <r+1:2> bits of the CWP because when one is incremented, the other is decremented by the same value. To show that this is true, let us consider a single window size (M). Thus, if X gives the nesting depth (0 for the first window, 1 for the second, and so on), then from the previous expressions we can deduce that the CWP and the CWRF for a function at level X are:

$CWP<31:2> = (2^{29} - 1) - X\,M,\ CWP<1:0> = 00$
$CWRF = (0 + X\,M) \bmod R = (X\,M) \bmod R$

*In reality the initial values of the CWP and the CWRF are different because initially the smallest window is allocated. However, we can assume that the initial values are the ones given above before allocating the window without any loss of generality.

If R is a power of two, then the <r+1:2> bits of the CWP differ from the r bits of the CWRF by the complementation constant $(2^{29} - 1)$, i.e., CWRF is the one's complement of CWP<r+1:2>.

R should be a power of two to implement these operations efficiently, i.e., to avoid divisions to compute the modulo and to use only two registers to manage the register file.

Note that, since we are using just one register for the CWP and the CWRF, after a context switching the top (largest) window for the new process (see Section 3.9) cannot be loaded at an arbitrary location, but in the one given by the one's complement of the <r+1:2> bits of the CWP.

4.2.2 Overflow and Underflow Detection

An overflow exception is detected when a new window is allocated or expanded and no more free registers are available in the register file. An underflow exception is detected when a function returns and the largest window size is not available in the register file. Since the callee does not know the window size of the caller, the largest window should be available (see Section 3.4). Let us discuss first how overflow and underflow conditions are detected and after this, how they are handled.

An overflow exception is detected when updating the CWRF we pass over the FWRF (see Figure 4.2), i.e., it is given by the following conditions:

$$\begin{array}{l} \textbf{if } CWRF \le FWRF \le (CWRF + Z_i) \textbf{ mod } R \textbf{ or} \\ \quad FWRF \le (CWRF + Z_i) \textbf{ mod } R < CWRF \textbf{ then} \\ \qquad \text{overflow exception} \\ \textbf{else } CWP \leftarrow CWP - Z_i,\ CWRF \leftarrow (CWRF + Z_i) \textbf{ mod } R \end{array}$$

This comparison is too difficult to implement. An alternative is to check that after decrementing the CWP, the number of registers given by the difference between the FWP and the CWP can be present in the register file, i.e.,

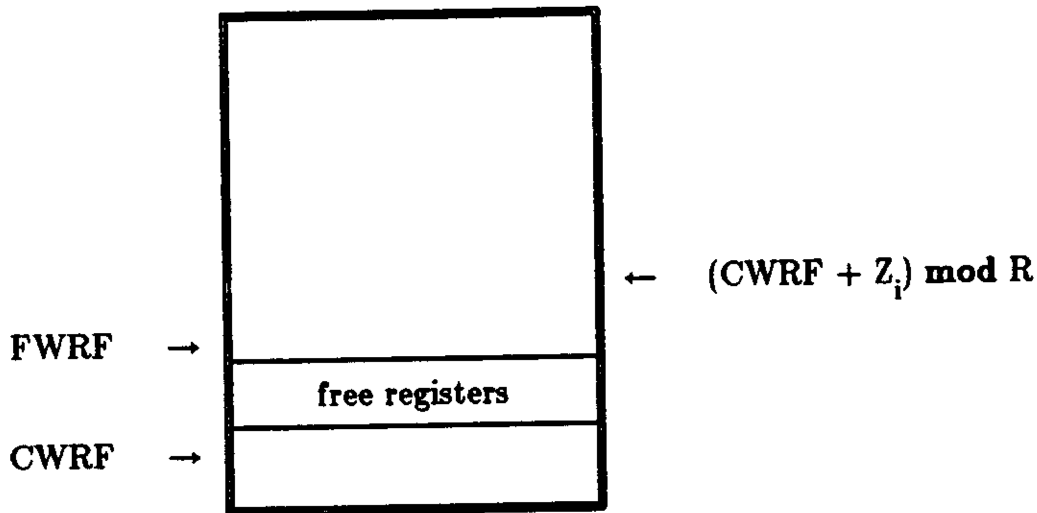
$$\begin{array}{l} CWP \leftarrow CWP - Z_i \\ \textbf{if } FWP - CWP \ge R \textbf{ then } \text{overflow exception} \end{array}$$

A 30-bit subtraction is not required because the high order bits of both pointers are always one as we said at the beginning of this section.

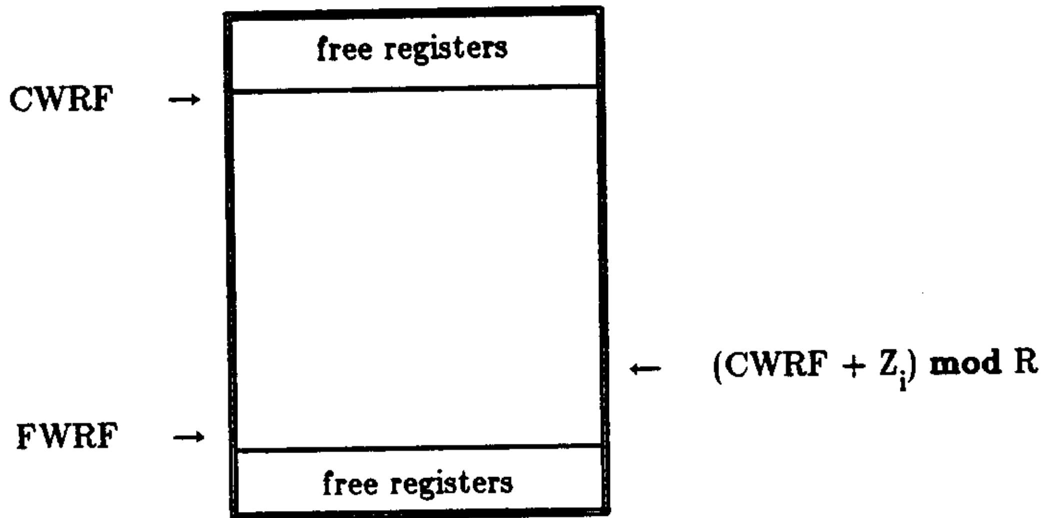
An underflow exception is detected when after updating the CWP, the largest window size is not available, i.e.,

$$\begin{array}{l} CWP \leftarrow CWP + M_i \\ \textbf{if } (CWP - FWP) < M_s \textbf{ then } \text{underflow exception} \end{array}$$

Again, a 30-bit subtraction is not required.



(a) Overflow if CWRF < FWRF ≤ (CWRF + Z_i) mod R



(b) Overflow if FWRF ≤ (CWRF + Z_i) mod R < CWRF

Figure 4.2. Register File Overflow

Both overflow and underflow conditions are handled by an operating system exception function or by the architecture itself depending on the complexity of the processor. Different strategies for managing the register file have been studied by Tamir and Séquin [TAMI83]. They conclude that with eight windows in the register file the best strategy is to bring only one window from or to memory. No new measurements have been taken for multi-size windows. We assume that on overflow (underflow) the largest window size is brought to (from) memory.

Thus, the following operations need to be performed:

(1) on overflow:

$$FWP \leftarrow FWP - M_s$$
$$FWRF \leftarrow (FWRF + M_s) \textbf{ mod } R$$

(2) on underflow:

$$FWP \leftarrow FWP + M_s$$
$$FWRF \leftarrow (FWRF - M_s) \textbf{ mod } R$$

Again, it can be shown that the register FWRF corresponds to the one's complement of the <r+1:2> bits of FWP if R is a power of two and their initial values are FWRF = (0,0,...,0) and FWP = (1,1,...,1,0,0).

Therefore, no extra complexity is introduced for multi-size windows to detect and to handle overflow and underflow exceptions, except that different constant numbers should be added/subtracted instead of a unique constant and that, since the window sizes are not necessarily a power of two, some more bits have to be added/subtracted (for a fixed-size window with 2^m registers, m bits of the CWP and the FWP are always zero so that they do not have to be added/subtracted).

4.2.3 Context Switching

When a new user process is scheduled, the windows associated to the old user process are saved in the window overflow stack area and only one large window associated to the new user process is brought to the register file, as we have discussed in Section 3.9. If registers are memory mapped, then an instruction (or a bit in the processor status word) should be provided to disable the register mapping. Special registers for global simple variables (see Section 2.10) have also to be saved during context switching, although this is not discussed here because is independent of the saving/loading of the register file. Let us discuss how context switching is handled for two different cases: two private register files, one for the user and the second shared among exception handlers (see Section 3.9.2), and a shared register file among the current user process and the exception handlers (see Section 3.9.3).

If two private register files are available, then the specialized registers (at least, the CWP and the FWP) can also be duplicated (one set for the user and a second for the kernel) so that they do not need to be saved when an exception occurs. As we said in Section 3.9.2, the user register file is not disturbed while the exception is handled.

If a new user process has to be scheduled as a result of the exception, then the user register file has to be saved in the user window overflow stack. Since the register file is organized as a random memory, registers are saved from the user register file to the memory locations of the user window overflow stack given by the values of the FWP and the CWP associated to the user process (not by the current FWP and CWP). The specialized registers have also to be saved in the user's process control block.

Special instructions should be provided to perform the context switching; however, these are not discussed in this thesis because they depend on the complexity of the processor. For instance, if the processor has only one-cycle instructions, then first the specialized registers are saved and after this, a special instruction saves the register pointed to by the FWP in the user window overflow stack, increments the user FWP, and sets a condition code when the user FWP is lower than the user CWP. This instruction is executed as many times as registers have to be saved. However, since a memory access is performed per instruction, two cycles are required to execute the instruction and to save only one register. Thus, an instruction to save a group of registers makes a better use of the parallelism in the processor, overlapping the access to the register file with the access to memory, so that the number of cycles required to perform the register saving is reduced. For instance, RISC uses the standard **stl** instruction to save one register, i.e., it takes two cycles (800 ns.) to save one register; while SOAR [UNGA84] provides one instruction to save eight registers in nine cycles.

Once the old user context is saved, the specialized registers and the larger window for the new user process are loaded. Again, some special instructions are required to perform these operations. Moreover, as we have said in Section 4.2.1, the window cannot be loaded in an arbitrary location in the user register file, but in the one given by the one's complement of the <r+1:2> bits of the CWP.

If the register file is shared among the user process and the exception handlers, then when an exception occurs (except an overflow or an underflow exception), a non-overlapping window is allocated. To do this, the CWP is decremented by (M_0 + P). However, before decrementing the CWP, this should be saved to remember where the last user window is located. To avoid the register saving, the CWP can also be duplicated. If the exception handlers and the user process are sharing a common window overflow stack, no duplication for the FWP is necessary. Otherwise, the FWP should also be duplicated (or saved) so that the overflow exception handler can differentiate user and kernel windows. For our discussion, let us assume that they share the overflow stack.

If a new user process has to be scheduled as a result of the exception, then the user's windows in register file have to be saved in the window overflow stack. To detect if the user still has any window present in the register file, the user CWP is compared with the current FWP. If the former is lower than the latter,

then this means that part of the register file (between the FWP and the user CWP) has to be saved. Otherwise, the user's windows have already been saved as a consequence of overflows generated during the execution of the exception handler functions. Although the exception handler might have some windows present in the register file, or part in the register file and part in the overflow stack, we assume that the scheduler never returns to its caller, i.e., it loads the new user's state, so that these windows are destroyed. Observe that the ones that are in the overflow stack are also destroyed because they have been saved above the user CWP. Likewise we have discussed above, special instructions are required to perform these operations.

Since the window cannot be loaded in an arbitrary location in the user register file, but in the one given by the one's complement of the <r+1:2> bits of the CWP, then it could be that these registers correspond to the ones used actually by the scheduler's window. Therefore, the scheduler should load the larger window at the end of its execution when it is possible to destroy whatever is in the registers. This is equivalent to what happens for a single-window architecture.

Finally, if we want to have always at least one window available to handle interrupts (as we said in Section 3.9.3), then we have to change the overflow detection algorithm given in the previous subsection so that an overflow for user processes is generated not when there are no more free registers, but when there is only one window left. For instance, to be able to allocate the smallest window size, the overflow detection algorithm for the processor user mode should be changed to:

$CWP \leftarrow CWP - Z_i$
if $FWP - CWP \geq (R - M_0 - P)$ **then** overflow exception

The operation $(R - M_0 - P)$ should not be performed because is known in advance.

Therefore, context switching is not more complex to manipulate in multi-size windows than in fixed-size windows because our discussion above can be applied either to fixed-size windows or to multi-size windows. Also, some special instructions have to be provided to perform the context saving and loading, independently of whether the current-window pointer and the first-window pointer are specialized registers.

4.2.4 Virtual to Physical Register Number Translation

Virtual register numbers (VR) given by the instruction have to be converted to physical register numbers (PR_ADDR) in the register file. In this subsection we study three different cases to see how costly the implementation can be for each case. In the first case, the number of registers in the fixed-size window is the same as the number of general-purpose registers, i.e., neither common nor overlapped registers are available (N = M). This case corresponds to the C/70 register file. In the second case, the register set consists of C common registers and a fixed-size window of M registers (N = C + M + P). This case corresponds to the RISC register file (with M = 16, C = 10, and P = 6), and to the PYRAMID 90x and CELERITY C1200 register files (M = 32, C = P = 16). In the third case, multi-size windows are considered with overlapped windows and no common registers.

(1) M = N

In this case, the register file is divided in W = R/N non-overlapped windows. W is a power of two because both N are R are powers of two. Translation is done by concatenation of the CWRF with the virtual register number, i.e.,

PR_ADDR ← (CWP<r+1:n+2>)' @ VR<n-1:0>

where the symbol ' represents the one's complement operator and the symbol @ represents the concatenate operator. CWP<1:0> is zero because the stack is word aligned and CWP<n+1:2> is zero because the window size (the register set) is a power of two.

(2) C ≠ 0 and M is a power of two

If there are common registers and the window size is a power of two, then half of the register set is used for common registers and outgoing-parameter registers and half for local registers and incoming-parameter registers (M = C + P = N/2). Let us consider that the common registers have the lower virtual register numbers (as they have in RISC and PYRAMID). In this case, a comparison is required to detect when a common register is used:

if VR < C **then** CR_ADDR ← VR<n-1:0>
else PR_ADDR ← (CWP<r+1:m+2>)' @ CC (VR<n-1:0>)

where CR_ADDR is the address of the common register file, CC is a code converted used to subtract C from the virtual register number, and *m* is the number of bits required to address a register in a window of size M.

If C is a power of two as is in PYRAMID (where C = P = N/4 = 16), then the translation mechanism is simplified:

```
if VR<n-1:n-2> = 00 then CR_ADDR ← VR<n-3:0>
else PR_ADDR ← (CWP<r+1:m+2>)' @ CC(VR<n-1:n-2>) @ VR<n-3:0>
```

(3) $C = 0$ and $M_0, M_1, \ldots, M_s$

With multi-size windows we assume that the outgoing-parameter registers have the lowest virtual register numbers so that the compiler always uses the same virtual register numbers for temporary registers and outgoing parameters. Thus, multi-size windows overlap as is shown in Figure 4.3. As we have said in Section 3.4, virtual register numbers that are not used in the current window are protected by the compiler, not by the architecture. Translation is performed using an r-bit adder:

```
PR_ADDR ← [ (CWP<r+1:2>)' - VR<n-1:0> ] mod R
```

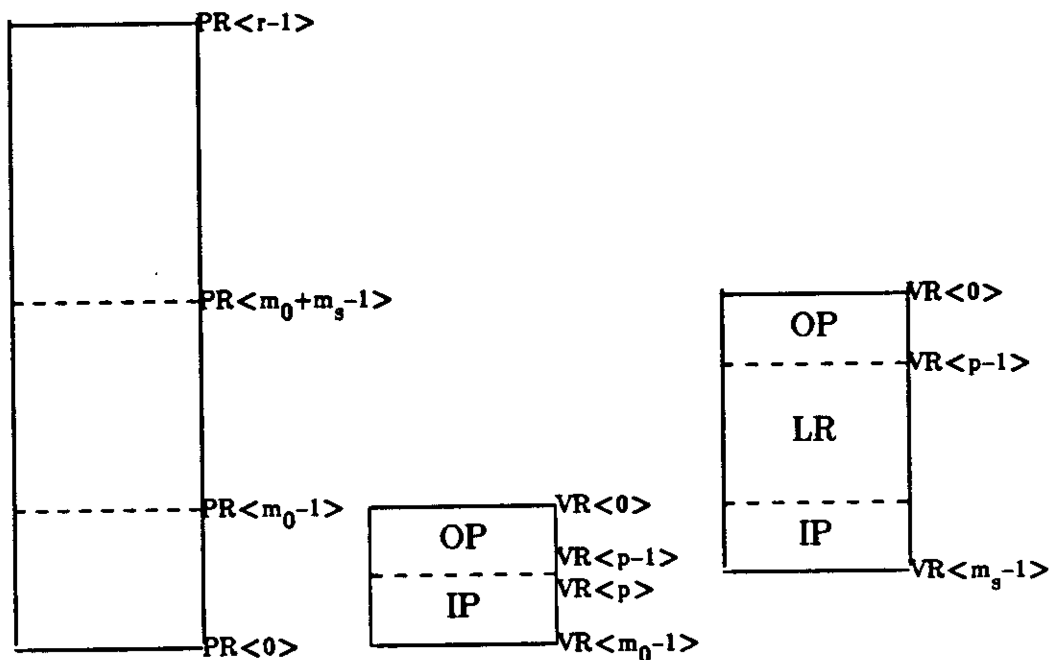


Figure 4.3. Mapping with Multi-Size Windows

In conclusion, if the window size is a non-power of two, then an r-bit adder is required to perform virtual register to physical register number translation. Since multi-size windows allow to have a small register file size, the number of bits to be added is small and the delay introduced is almost insignificant. This number is even reduced in a shift-register file as we will discuss in Section 4.5. Therefore, although in C/70, RISC, PYRAMID, and CELERITY the translation is

performed with concatenation and this is for sure a faster operation than addition, the number of bits required to be added does not force us to discard the use of non-power-of-two windows. Furthermore, translation is simplified when no common registers are available.

4.2.5 Memory Mapped Registers

Although we have already seen in Section 2.3 that the alias problem can be solved by an optimizing compiler, here we would like to study the complexity that must be introduced to the processor when registers are memory mapped. To detect if a memory address (MA) location is in a register instead of memory, the MA must be compared with the the current and first window pointers:

$$CWP \leq MA \leq FWP$$

Once it is detected that the memory address refers to a register the physical register number has to be computed. This computation is performed subtracting the MA from the FWP to get the register number relative to the register number pointed by FWRF. The physical register number is obtained adding the FWRF and the result of the subtraction modulo R:

$$\begin{aligned}
&[\, FWRF + (FWP{<}31{:}2{>} - MA{<}31{:}2{>}) \,]\ \mathbf{mod}\ R = \\
&[\, FWRF\ \mathbf{mod}\ R + (FWP{<}31{:}2{>} - MA{<}31{:}2{>})\ \mathbf{mod}\ R \,]\ \mathbf{mod}\ R = \\
&[\, FWRF\ \mathbf{mod}\ R + FWP{<}31{:}2{>}\ \mathbf{mod}\ R - MA{<}31{:}2{>}\ \mathbf{mod}\ R \,]\ \mathbf{mod}\ R
\end{aligned}$$

If R is a power of two, then $FWRF = (FWP{<}r{+}1{:}2{>})'$ and

$$\begin{aligned}
&[\, (FWP{<}r{+}1{:}2{>})' + FWP{<}r{+}1{:}2{>} - MA{<}r{+}1{:}2{>} \,]\ \mathbf{mod}\ R = \\
&[\, (2^r - 1) - MA{<}r{+}1{:}2{>} \,]\ \mathbf{mod}\ R = \\
&[\, (MA{<}r{+}1{:}2{>})' \,]\ \mathbf{mod}\ R = \\
&(MA{<}r{+}1{:}2{>})'
\end{aligned}$$

Therefore, if R is a power of two, r bits of the memory address give directly the physical register number independently of the window size (power or non-power of two). With a circular-buffer organization, the complexity introduced by memory mapped registers consists basically in detecting when a memory address refers to a registers, not in translating the memory address to a physical register number.

In conclusion, in a register file with a circular-buffer organization the number of registers in the register file must be a power of two to be able to implement efficiently the operations on both the current-window pointer and the first-window pointer. However, the window size does not have to be necessarily a power of two. When the window size is not a power of two, an r-bit addition needs to be performed to translate virtual to physical register numbers.

4.3 Shift-Register File

This section discusses an alternative implementation to the circular-buffer register file, called the ***shift-register file***. The name comes from the fact that the stack of windows is organized as a shift register as shown in Figure 4.4. Read and write operations can only be performed on the registers of the top window. To introduce how the register file is organized with shift registers we have selected to start with non-overlapped fixed-size windows because it is easier to explain; later, Sections 4.4 and 4.5 show how the shift-register file can be organized to support fixed-size window with overlapped registers and multi-size windows. In this section, we also present the advantages of the shift-register file with respect to the circular-buffer one.

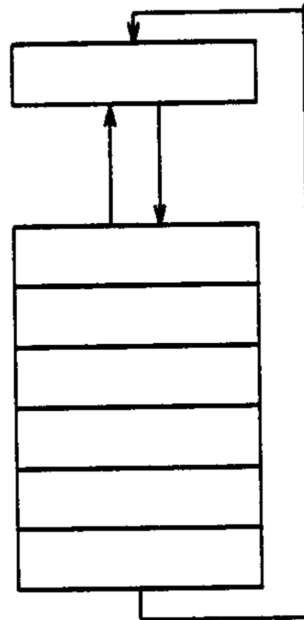


Figure 4.4. Shift-Register File with Fixed-Size Windows

If the registers are not memory mapped, then only access to the current window is required, i.e., we do not need access to the whole register file. Thus, the register file is implemented as it is shown in Figure 4.4 so that registers are pushed (shifted down) when a function is called and poped (shifted up) when a function returns. In this case, a bus of the same size as in the single window case is required so that both the bus size and the access time to the register file depends only on the register set, and not on the number of windows.

The shift-register file is implemented as a circular shift register because when the file is full and a new function is called the bottom register window becomes available in the top one so that an exception handler (an operating system function or the architecture itself as we have already discussed in the previous section) can transfer it to memory. To detect the overflow a bit is associated to

each window indicating when the window is in use; this bit is shifted up/down together with the window. When a function is called and the register file is shifted down, an overflow is detected if the bit associated to the top window indicates that the window is in use. As in the circular-buffer case, the first-window pointer indicates the memory address where the window must be saved.

Underflow is detected using the same bit used for overflow detection. When a function returns and the register file is shifted up, an underflow is detected if the bit associated to the top window indicates that the window is not in use. Again, the FWP is used to transfer a window from memory to the top window.

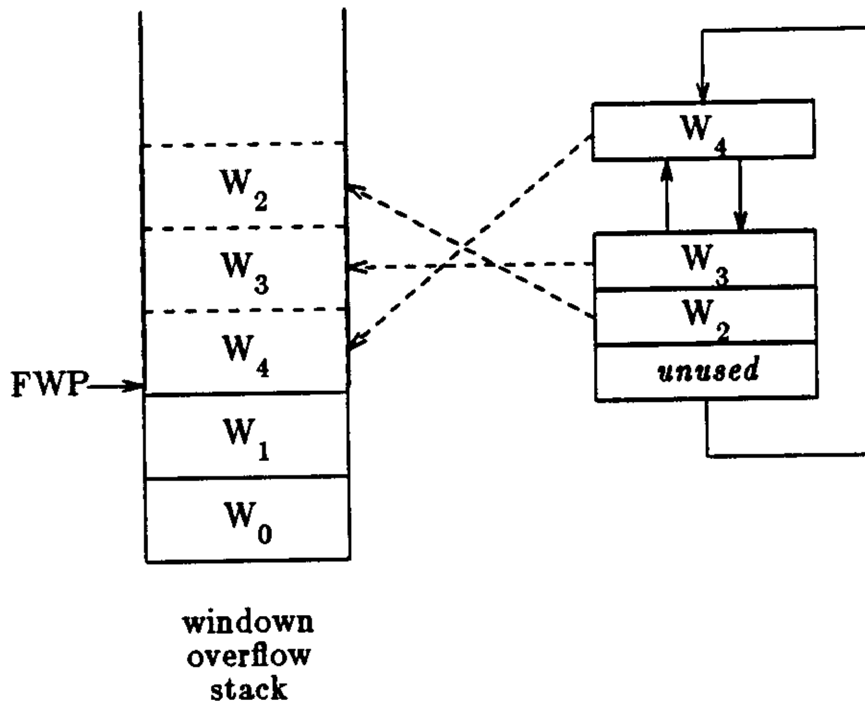
Although in this case the current-window pointer is not necessary to perform translation from virtual to physical register numbers because this is direct (i.e., the same virtual register numbers are used as physical register numbers), it is still necessary to handle context switching. Let us discuss why.

If only the FWP is available, then windows have to be saved from the top to the bottom because only access to the top window is provided. Thus, windows in the overflow stack are saved in the opposite order (see Figure 4.5.(a)) and a mask has to be saved on the top of the overflow stack indicating how many windows have been saved. When this process is scheduled to run again, the processor state has to be restored completely because the top windows in the stack are upside down. As we said in Section 3.9, this is not a good policy because if the process is performing more function calls than returns, some windows would have to be transferred again to memory without having been used.

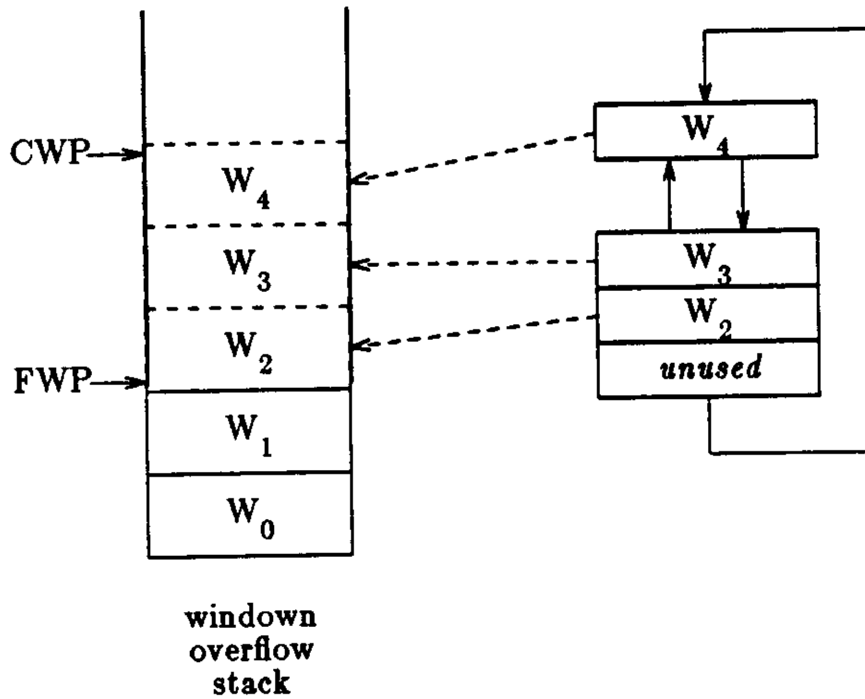
If the CWP is also available, then the top window in the register file is saved at the top of the overflow stack, the next window is saved below the previous one, and so on (see Figure 4.5.(b)). In this case, when this process is scheduled to run again, only the top window is loaded in the register file. This policy corresponds to the one discussed in Section 3.9 and implemented for a circular-buffer register file.

Thus, the CWP is required to handle context switching. Besides the order in which windows are saved, context switching is handled similarly to the circular-buffer case (see Section 4.2.3).

The register window size does not have to be necessarily the same as the number of registers in the register set. Some registers are used as temporary registers and, therefore, they do not need to be pushed when a function is called. Thus, the register file does not have to be a multiple of the register set, and the common registers can be used to pass parameters as has been discussed in Section 3.5 for non-overlapped windows.



(a) Using the FWP



(b) Using the CWP

Figure 4.5. Saving the Register File on Context Switching

The shift-register file offers the following advantages as compared to a circular-buffer register file:

(1) The size of the register cells (except the ones which are on the top) might be smaller because no bus access is required for them. However, control to perform the up/down shifting is required. A detailed design is being performed to evaluate the reduction in size that can be obtained. The percentage of the chip area dedicated to register storage is 39% for RISC I (gold implementation*), 30% for RISC I (blue implementation) [FITZ82], and 27.5% for RISC II [KATE83].

(2) Address decoders are only needed for the top window reducing the chip area required for this. The percentage of the chip area dedicated to decoders is 8% for RISC I (gold implementation), 6% for RISC I (blue implementation), and 5.8% for RISC II.

(3) The processor cycle time is reduced because:

(3a) The mapping from virtual to physical register number is direct. So, the time required to perform the addition to compute the physical register number for non-power-of-two windows is eliminated. No addition is required for power-of-two windows (see Section 4.2.4).

(3b) The number of registers available to perform reads and writes is smaller so that the bus size is also smaller. Thus, the access time to the register file is similar to the single window case and it is independent of the size of the register file. This eliminates one of the drawbacks for using multiple windows given in [HENN84] and [PATT85].

(4) The number of registers available in the register file does not have to be a power of two. This allows a more flexible selection of the number of registers required in the file. In the circular-buffer register file the number of registers in the file must be a power of two (see Section 4.2.1).

On the other hand, the shift-register file offers two disadvantages with respect to a circular-buffer register file when it is shared among the user process and the exception handlers:

(1) The overflow detection algorithm for the user process cannot be modified so that there exists always at least one free window to handle exceptions. This is because only access to the top window is provided. Section 4.2.3 has shown how the detection algorithm can be modified for the circular-buffer case.

*RISC I was implemented by two different groups: one was called the gold group and the second the blue group.

(2) When a new user process is scheduled, the kernel windows left in the register file have to be destroyed (shifted up) so that the user windows are moved to the top to be transferred to the overflow stack (if any). Since access to the whole register file is provided in a circular-buffer organization, the user windows can be transferred directly to memory.

The implementation of the shift-register file is not discussed in this thesis. Marc Tremblay, a graduate student at UCLA, is working actually on it. However, only a remark is done to the reader. To avoid to have a very long connection from the bottom of the file to the top, windows can be physically reordered so that all the connections become of equal size. Figure 4.6 shows how individual bit cells might be connected to optimize the connection length for seven windows numbered from 1 to 7; window 1 is the top window and window 7 is the bottom one.

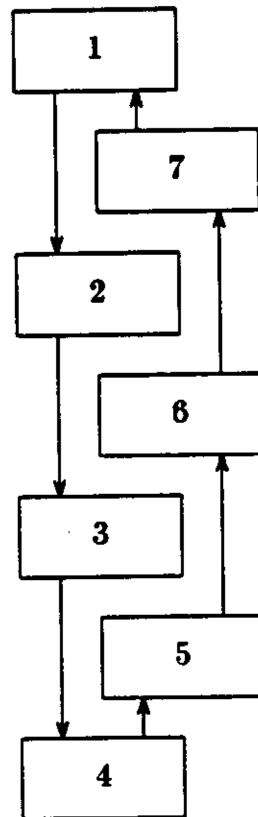


Figure 4.6. Cell Connections in the Shift-Register File

In conclusion, a shift-register file allows a more flexible selection of the number of registers required in the file, makes the register file access time similar to the one required for the single window case, and we expect that less area will be required for its implementation.

4.4 Overlapped Windows in a Shift-Register File

This section discusses how the shift-register file is organized to implement fixed-size windows with overlapped registers. The window is divided into two parts: the overlapped registers and the local ones. The overlapped registers are also divided into two parts: the registers for incoming parameters (IP) and the registers for outgoing parameters (OP) used also as temporary registers. The only difference with the overlapped windows in the circular-buffer register file is that here both the IP and the OP registers have the lowest virtual register numbers while in the circular-buffer approach the IP and the OP register numbers are in opposite sides of the register set (if no common registers are considered). Let us call the part of the register file with the local registers the local-shift-register file (LSRF) and the other part the overlapped-shift-register file (OSRF). Figure 4.7 shows the new structure for the shift-register file.

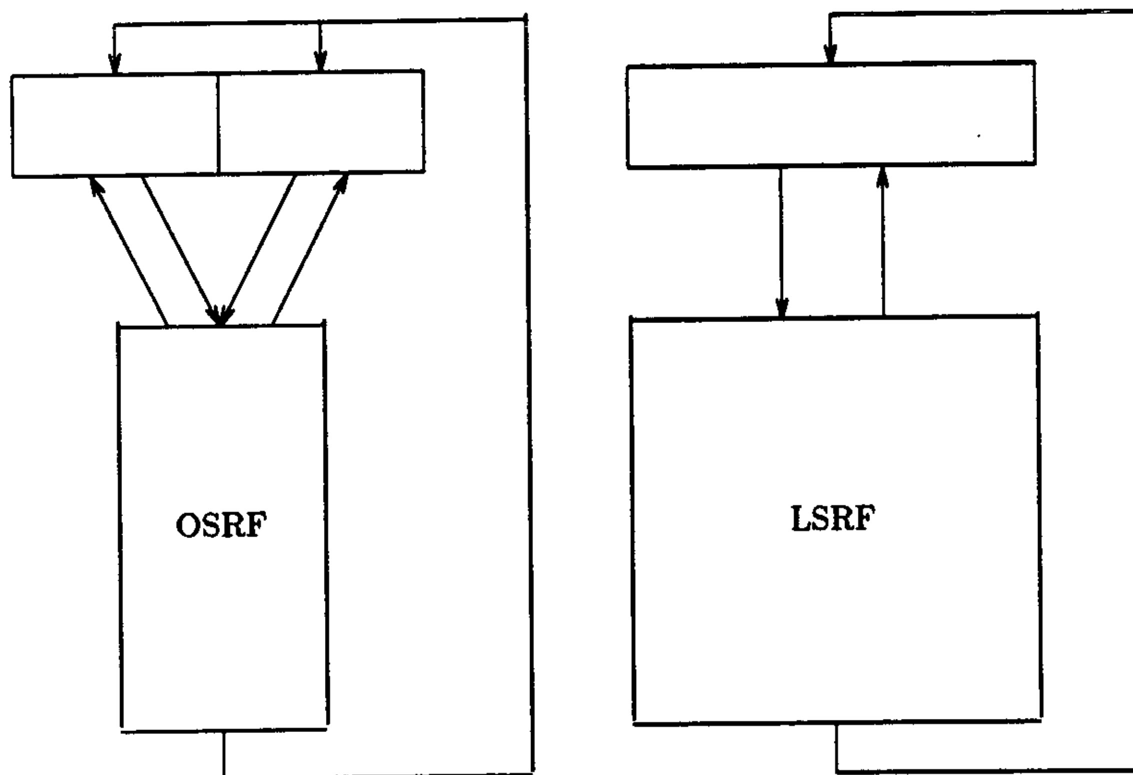


Figure 4.7. Shift-Register File with Overlapped Windows

Let us discuss the operations to be performed on function call and return, on translating virtual to physical register numbers, on overflow and underflow, and on context switching.

Window Allocation and Deallocation

When a function is called the local-shift-register file is shifted down as we said in the previous section, but for the overlapped-shift-register file only the IP registers have to be shifted down. The OP registers have to remain in the top window to be accessed as the IP registers for the callee function.

When a function returns, the callee IP registers must remain in the top window because they contain the return value and they are the caller OP. Thus, only the caller IP registers and the local-shift-register file have to be shifted up.

Therefore, a multiplexer is needed to select the part of the overlapped registers that are shifted up or down. How the overlapping is performed is explained next.

Virtual to Physical Register Translation

The mapping from virtual to physical register numbers is still direct for the local-shift-register file. However, the mapping for the overlapped-shift-register file is not direct because the OP registers of the caller must become the IP registers of the callee. Thus, the mapping must alternate the translation from virtual to physical register numbers so that the same physical registers used by the caller as OP are used by the callee as IP.

To obtain this alternative mapping, functions are classified in *even* and *odd* functions depending on their stack (or window) depth in the moment they are called, i.e., the main function is an *even* function, functions called by the main function are *odd* functions, function called by these become *even* again, and so on. The <m+2> bit of the CWP can be used to detect when a function is *even* or *odd*. Thus, for one type of functions (let us say *even*), the mapping is direct:

$$VR_0 \rightarrow PR_0, VR_1 \rightarrow PR_1, ..., VR_{2P-1} \rightarrow PR_{2P-1}.$$

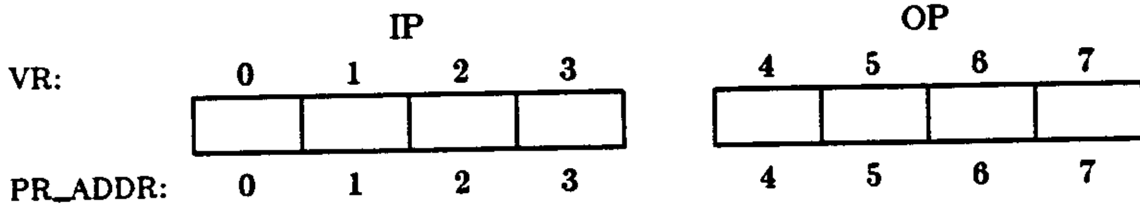
For the other type of functions, the mapping is the opposite:

$$VR_0 \rightarrow PR_{2P-1}, VR_1 \rightarrow PR_{2P-2}, ..., VR_{2P-1} \rightarrow PR_0.$$

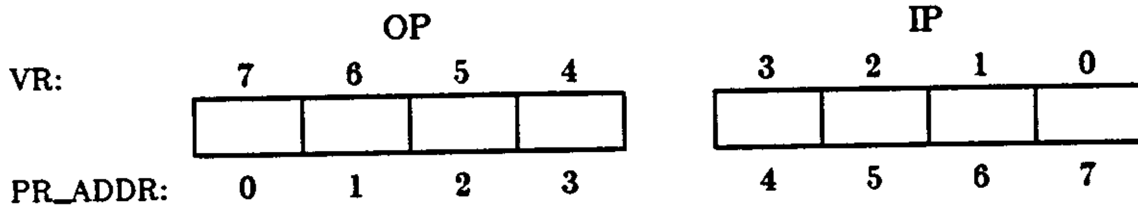
If the number of overlapped registers is a power of two, then the physical register number for the *odd* functions is just given by the one's complement of the virtual register number. Otherwise, a code converter is necessary.

The mapping from virtual to physical register numbers is shown in Figure 4.8 for four overlapped registers. Observe that the mapping is transparent to the compiler. The compiler has to know only the register numbers used as IP and OP registers and their relationship (VR_0 contains the return address, the caller must

use register VR_6 to pass the first parameter to the callee, which receives in its register VR_1, etc.). The compiler has to know also this information for the circular-buffer case.



(a) *even functions*



(a) *odd functions*

Figure 4.8. Sharing Four Overlapped Registers

Overflow and Underflow Handling

Overflow and underflow conditions are detected using a bit associated to each window to indicate when a window is in use as has been discussed in the previous section.

When a function is called, the register file is shifted down. The bottom window of the local-shift-register file is moved to the top one, and the bottom window of the overlapped-shift-register file is moved to the OP part of the top window. If an overflow is detected, then the local registers and the OP registers are transferred to memory.

When a function returns, the register file is shifted up. The IP registers of the callee are converted to the OP registers of its caller so that they are always present. If an underflow is detected, then the local registers and the OP registers are brought from memory.

Context Switching

To switch the context from one user to another we perform the same operations described in the previous section, i.e., we use the CWP to move windows in the register file (from the top to the bottom) to the overflow stack. The first time, the whole top window is moved; for the next ones, only the local registers and the IP registers are moved.

The two drawbacks mentioned in the previous section for a shared shift-register file with respect to the circular-buffer case are still valid. However, now a third one appears: since only half of the overlapped registers can be shifted down, it is impossible to allocate a non-overlapped window. In this case, the easiest solution is to trust that the kernel functions will not use the incoming-parameter registers (of course, this only affects to the first function).

A Comparison with the Previous Approach

Overlapped windows introduce two drawbacks as compared to non-overlapped windows:

(1) Some extra area is required for the multiplexer and control introduced by the overlapped-shift-register file.

(2) Virtual to physical register number translation is required for overlapped registers. However, if the number of overlapped registers is a power of two, the complexity required by the translation (i.e., to compute the one's complement of the virtual register number) is insignificant.

The advantages of having overlapped register windows have already been discussed (see Section 3.5) and compensate these drawbacks. Therefore, overlapped windows can also be implemented using a shift-register file. Moreover, if we compare overlapped registers here with the circular-buffer case, then another advantage appears: the total number of registers in the file can be used in full. Remember that in the circular-buffer organization the outgoing-parameter registers of the last window are overlapped with the incoming-parameter registers of the first window so that registers associated to the last window cannot be used. In the shift-register file this restriction disappears. However, the sharing of the register file by the user process and the exceptions handlers is better protected with a circular-buffer organization than with a shift-register organization because in the latter a non-overlapped window cannot be allocated.

4.5 Multi-Size Windows in a Shift-Register File

This section discusses how multi-size overlapped register windows can be implemented using a shift-register file. The top window corresponds to the largest window size plus the incoming parameter registers, i.e., if no common registers are available, then the top window corresponds to the largest general-purpose register set (of size N). The register file is divided in N/P groups of P registers. Each group of P registers is shifted up/down individually and each level in the file has associated a bit to detect whether the group is in use. This bit is equivalent to the one we had for each window in the previous two cases to detect overflow/underflow conditions. The window sizes available are: P, 2P, 3P, ..., and N–P because a whole group of registers have to be shifted down (allocated) or shifted up (deallocated) as we will discuss below.

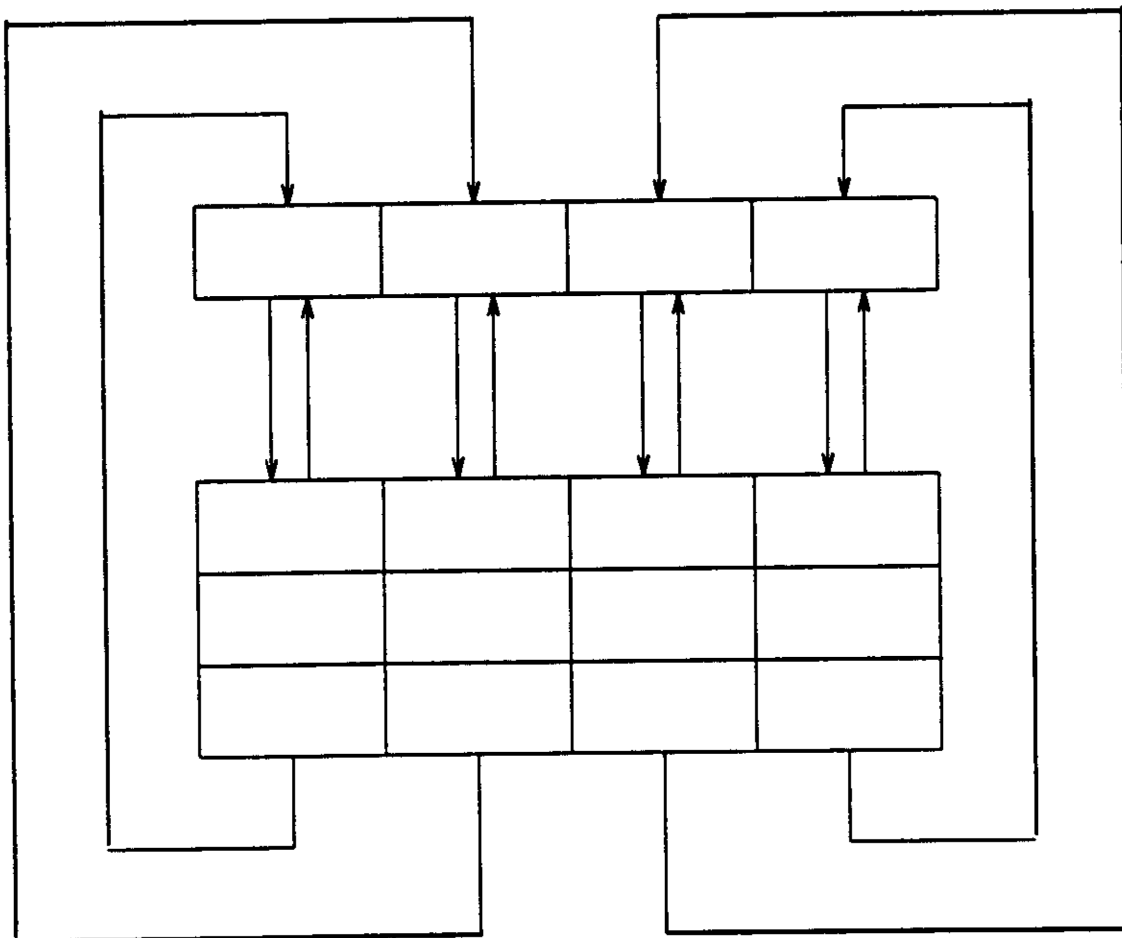


Figure 4.9. Shift-Register File with Multi-Size Windows

For instance, Figure 4.9 shows a shift-register file divided in four groups. If every group has four registers (P = 4), this file corresponds to a general-purpose register set with 16 registers, four overlapped registers, window sizes of 4, 8, and 12 registers, and a total of 64 registers in the file.

Let us discuss again the operations to be performed on function call and return, on translating virtual to physical register numbers, on overflow and underflow, and on context switching.

Window Allocation and Deallocation

Since now windows are of different sizes, we need a pointer to the top of the window similarly to the CWRF pointer discussed for the circular-buffer case (see Section 4.2). To smallest window size that it is possible to allocate is $M_0 = P$ because a minimum of P registers have to be shifted down. Thus, when a function is called, this pointer is incremented by the smallest window size (P) modulo N. The successive window sizes must be multiples of P because P registers are shifted up/down all together.

If the callee requires a larger window, then an instruction generated by the compiler will update the CWP and shift down one or more groups. When the function returns, the window is deallocated so that the groups for such window are shifted up. Thus, the operations to be performed are very similar to the ones described for a circular-buffer organization, except that they are done modulo N instead of modulo R:

(1) on function call:

$$CWP \leftarrow CWP - M_0$$
$$CWRF \leftarrow (CWRF + M_0) \textbf{ mod } N$$

(2) on expanding the window size up to M_i:

$$CWP \leftarrow CWP - Z_i$$
$$CWRF \leftarrow (CWRF + Z_i) \textbf{ mod } N$$

(3) on function return:

$$CWP \leftarrow CWP + M_i$$
$$CWRF \leftarrow (CWRF - M_i) \textbf{ mod } N$$

Also, it can be shown that this pointer corresponds to the one's complement of the bits <n+1:2> of the CWP, or if P is a power of two, to the one's complement of the bits <n+1:p+2> of the CWP (see Section 4.2.1).

Virtual to Physical Register Translation

A shift-register file with multi-size windows has the overlapped registers in the opposite sides of the current window as the circular-buffer case. Virtual registers are also numbered as the circular-buffer case, i.e., the outgoing-parameter registers have the lowest virtual register numbers (see Figure 4.3). The translation from virtual to physical register numbers is performed similarly to the circular-buffer case (see Section 4.2.4), but only an n-bit adder is required:

$$PR_ADDR \leftarrow [\,(CWP<n+1:2>)' - VR<n-1:0>\,]\ \mathbf{mod}\ N$$

If P is a power of two, then it is only necessary to use a (n-p)-adder:

$$PR_ADDR \leftarrow [(CWP<n+1:p+2>)' - VR<n-1:p>]\ \mathbf{mod}\ N\ @\ VR<p-1:0>$$

Thus, for instance, to translate virtual to physical register numbers for a 64-register file with three-size windows of sizes 4, 8, and 12, and 4 overlapped registers, a 2-bit adder is required for a shift-register organization, while a 6-bit adder is required for a circular-buffer organization.

Overflow and Underflow Handling

An overflow exception is detected when a group of registers is shifted down and the bit associated to the top window for this group indicates that the registers are in use. An underflow exception is detected similarly.

The main difference with the circular-buffer organization is that the largest window cannot be any longer transferred to/from memory when overflow/underflow is detected. With a shift-register organization, only the groups that have caused overflow/underflow can be transferred. On overflow, other groups cannot be transferred to memory because they are at the bottom of the register file so that they are not available. On underflow, other groups cannot be brought from memory because the other groups at the top window are in use.

Therefore, only the exact number of registers which have caused the overflow/underflow condition are transferred to/from memory. Although no measurements have been taken, it seems that this results in a slightly greater percentage of overflows because when the process is increasing its stack depth, two overflows can be generated by the same function (one for the smallest window size and a second if the function requires a larger window size).

Context Switching

Context switching is handled similarly how has been discussed in Section 4.3. Windows are saved, from the top to the bottom, to the overflow stack using the CWP. When the process is restored, then the whole top window cannot be loaded to an arbitrary location. It has to be loaded in the order that the <n+1:2> (or <n+1:p+2> if P is a power of two) bits of the CWP specifies because virtual register numbers are translated relatively to the contents of these bits.

The two drawbacks of the shared shift-register file with respect to the shared circular-buffer case mentioned in Section 4.3 remain. However, in this case, when an exception occurs, it is possible to allocate a non-overlapped window with the smallest size shifting down two groups of registers. Thus, the third drawback introduced in the previous section for fixed-size windows with overlapped registers disappears.

In conclusion, multi-size windows can also be implemented using a shift-register file. The advantages with respect to a circular-buffer organization is that a shift-register file makes the register file access time independent of the register file size, performs virtual to physical register number translation using an n-bit adder or a (n-p)-adder if P is a power of two, and allows more flexibility in the selection of the register file size because R should be just a multiple of N. The drawbacks are that it forces the window sizes to be a multiple of P, and that, although no measurements have been taken, it seems that it generates a slightly greater percentage of overflows because only the exact number of registers can be transferred to memory.

4.6 Conclusions

In this chapter we have discussed two different organizations for the register file: circular buffer and shift registers. The circular-buffer organization has already been used for fixed-size windows in the RISC and PYRAMID 90x processors. We have seen how it can also be used to implement multi-size windows and have shown that the window size does not have to be necessarily a power of two. If the window size is not a power of two, then an r-bit adder is required to perform translation from virtual to physical register numbers.

A new organization called the shift-register file has been introduced. We have explained how the shift-register file can also be used to implement fixed-size windows without and with overlapped registers, and multi-size windows. The advantages of using a shift-register organization rather than a circular-buffer one are:

(1) The register file access time depends only on the size of the general-purpose register set and not on the size of the register file. Thus, the processor cycle delay caused by access to the register file is independent of the register file size and similar to the delay required to access a register file with a single window.

(2) A shift-register file allows more flexibility in the selection of the size of the file: R should only be a multiple of N. In the circular-buffer register file R should be a power of two.

(3) To translate a virtual register number to a physical register number an n-bit adder is required instead of an r-bit adder. The number of bits to be added can be reduced to (n-p) if P is a power of two. For instance, to translate virtual to physical register numbers for a 64-register file with three-size windows of sizes 4, 8, and 12, and 4 overlapped registers, a 2-bit adder is required for a shift-register organization, while a 6-bit adder is required for a circular-buffer organization.

(4) All the registers in the shift-register file can be used. In the circular-buffer register file, the last window cannot be used because its outgoing-parameter registers overlapped with the incoming-parameter registers of the first window.

We also expect a reduction in the chip area required for storage, bus, and decoders. However, no results are available yet.

CHAPTER 5
CONCLUSIONS

In this thesis, the architecture of a register file for a processor oriented to execute programs written in C has been considered to study the cost/performance tradeoff for the register set design. The principle that has motivated Reduced Instruction Set Computers, to utilize the resources efficiently to support the execution of the most frequent instructions, has been the main motivation for this design.

In Chapter 2, we have discussed the use of the registers made by the compiler for different purposes: for local simple variables defined by the programmer, for temporary results and parameter passing, for optimizing variables defined by the compiler, for the run-time process environment, and for global simple variables. The problems that a compiler has to use the registers for each purpose have been considered and the solutions offered by an optimizing compiler and by the architecture have been evaluated. We have concluded that registers can be used by an optimizing compiler to reduce both the data memory traffic and the instruction memory traffic, even if only a single window is available.

In Chapter 3, we have confirmed that multiple windows offer an excellent support for function calls independently of the instruction set design (reduced or complex). The advantages and disadvantages of fixed-size and variable-size windows have been studied and multi-size windows have been proposed as an intermediate solution that incorporates the advantages of both schemes. Using our measurements we have concluded that three-size windows with four overlapped registers and window sizes of 4, 8, and 12, and a 64-register file balance the cost/performance tradeoff.

The effects of multiprogramming on the register file have also been considered and we have shown that (1) the register file should not be saved totally when an exception occurs because of the high frequency of exceptions and the low average service time, and (2) the register file is fully utilized because the average number of function calls between context switches is greater than the average number of function calls between overflows. We have also concluded that with a private organization, two register files are required one for the user processes and the other for the exception handlers, and that with a shared organization, the register file should be shared only among the user and the exception handlers, not among several user processes.

In Chapter 4, we have shown how a circular-buffer register file can also be used to implement multi-size windows and have compared this implementation with the existing organizations with fixed-size windows. Moreover, a new organization has been proposed: the shift-register file. The shift-register file offers the following advantages with respect to the circular-buffer register file: it makes the bus size and the access time to the register file independent of its size, it allows more flexibility in the selection of the size of the file, it requires a smaller adder to perform virtual to physical register number translation, it utilizes all the registers available in the file, and it might reduce the required area.

Therefore, the use of an optimizing compiler and a multiple-window register file with multi-size windows balances the cost/performance tradeoff of the register set design: it allows an excellent usage of the register file, it increases the locality of the references, and it reduces both the data and the instruction memory traffic generated.

References

[AHO78] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts (1978).

[ALEX75] W. Gregg Alexander and David B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer* **8**(11), pp.41-6 (November 1975).

[ANKL82] P. Anklam, D. Cutler, R. Heinen, Jr., and M. D. MacLaren, *Engineering a Compiler*, Digital Equipment Corporation, Bedford, Massachusetts 01730 (1982).

[AUSL82] M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp.22-31 (June 1982). Published in *SIGPLAN Notices* **17**(6).

[BASA83] Ed Basart and Dave Folger, "RIDGE 32 Architecture - A RISC Variation," *IEEE International Conference on Computer Design: VLSI in Computers*, pp.315-8 (November 1983).

[BBN81] *C/70 Hardware Reference Manual*, BBN Computer Company, Cambridge, MA (1981).

[BEAT78] J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM Journal of Research and Development* **18**(1), pp.20-39 (January 1978).

[BEEL84] Michael Beeler, "Beyond the Baskett Benchmark," *Computer Architecture News* **12**(1), pp.20-31 (March 1984).

[BERE82] A. Berenbaum, M. Condry, and P. Lu, "The Operating System and Language Support Features of the BELLMAC-32 Microprocessor," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.30-8 (March 1982). Published in *Computer Architecture News* **10**(2).

[BEST82] D. W. Best, C. E. Kress, N. M. Mykris, J. D. Russell, and W. J. Smith, "An Advanced-Architecture CMOS/SOS Microprocessor," *IEEE Micro* **2**(3), pp.11-26 (August 1982).

[CHAI81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," *Computer Languages* **6**, pp.47-57 (1981).

[CHAI82] G. J. Chaitin, "Register Allocation & Spilling Via Graph Coloring," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp.98-105 (June 1982). Published in *SIGPLAN Notices* **17**(6).

[CHOW84] F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp.222-32 (June 1984). Published in *SIGPLAN Notices* **19**(6).

[CLAR82] D. W. Clark and H. M. Levy, "Measurement and Analysis of Instruction use in the VAX-11/780," *Proc. of the 9th Symposium on Computer Architecture*, pp.9-17 (1982).

[COOK82] R. P. Cook and I. Lee, "A Contextual Analysis of Pascal Programs," *Software Practice & Experience* **12**(2), pp.195-203 (February 1982).

[DANN79] Roger B. Dannenberg, "An Architecture with Many Operand Registers to Efficiently Execute Block-Structured Languages," *Proceedings of the 6th Annual Symposium on Computer Architecture*, pp.50-7 (April 1979).

[DEC77] *11/60 Microprogramming Specification*, Digital Equipment Corporation, Maynard, MA (1977).

[DEC78] *PDP-11 Processor Handbook*, Digital Equipment Corporation (1978).

[DEC79] *VAX-11 Architecture Handbook*, Digital Equipment Corporation (1979).

[DEC80] *VAX Hardware Handbook*, Digital Equipment Corporation (1980).

[DITZ82] D. R. Ditzel and H. R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.48-56 (March 1982). Published in *Computer Architecture News* **10**(2).

[EMER84] J. S. Emer and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *The 11th Annual International Symposium on Computer Architecture*, pp.301-10 (June 1984). Published in *Computer Architecture News* **12**(3).

[FEUE82] A. R. Feuer and N. H. Gehani, "A Comparison of the Programming Languages C and PASCAL," *ACM Computing Surveys* **14**(1), pp.73-92 (March 1982).

[FITZ82] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, and K.S. Van Dyke, "A RISCy Approach to VLSI," *Computer Architecture News* **10**(1), pp.28-32 (March 1982).

[FREI74] R. A. Freiburghouse, "Register Allocation Via Usage Counts," *Communications of the ACM* **17**(11), pp.638-642 (November 1974).

[GAJE75] Hania Gajewska, "Some statistics on the Usage of the C Language," Technical Memorandum 75-1273-13, Bell Laboratories, Murray Hill, New Jersey 07974 (November 24, 1975).

[GILB81] Jim Gilbreath, "A High-Level Language Benchmark," *BYTE* **6**(9), pp.180-98 (September 1981).

[GRAP81] R. D. Grappel and J. E. Hemmenway, "A Tale of four μPs: Benchmarks quantify performance," *EDN* **26**(7), pp.179-265 (April 1, 1981).

[GROS83] Thomas Gross and John Gill, "A Short Guide to MIPS Assembly Instructions," Technical Note No. 83-236, Computer Systems Laboratory, Stanford University, Stanford, CA 94305 (November 1983).

[HARB82] S. Harbison, "An Architectural Alternative to Optimizing Compilers," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.57-65 (March 1982). Published in *Computer Architecture News* **10**(2).

[HENN82] J. Hennessy, N. Jouppi, J. Gill, R. Baskett, A. Strong, T. Gross, C. Rowen, and J. Leonard, "The MIPS Machine," *COMPCON 82*, pp.2-7 (Spring 1982).

[HENN84] John L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers* **C-33**(12), pp.1221-46 (December 1984).

[HILL83] D. D. Hill, "An Analysis of C Machine Support for Other Block-Structured Languages," *Computer Architecture News* **11**(4), pp.7-16 (September 1983).

[HOR85] T. M. Hor and C. K. Yuen, "The Design and Programming of a Powerful Short Wordlength Processor Using Context-Dependent Machine Instructions," *Computer Architecture News* **13**(1), pp.12-26 (March 1985).

[HUCK83] Jerome C. Huck, "Comparative Analysis of Computer Architectures," Technical Report No. 83-243, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA. 94305 (May 1983).

[HUGU81] Miquel Huguet, "Monitorització del Sistema Operatiu RSX-11M," Minor Thesis, Facultat d'Informàtica, Universitat Politècnica, Barcelona, Catalonia (December 1981).

[JOHN79] S. C. Johnson, "A Tour Through the Portable C Compiler," in *UNIX Programmer's Manual for Advanced Programmers*, Bell Telephone Laboratories, Murray Hill, New Jersey 07974 (January 1979).

[JOHN81] S. C. Johnson, "Position Paper on Optimizing Compilers," *8th Annual ACM Symposium on Principles of Programming Languages*, pp.88-9 (January 1981).

[JOHN82] R. Johnsson and J. Wick, "An Overview of the Mesa Processor Architecture," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.20-9 (March 1982). Published in *Computer Architecture News* **10**(2).

[KATE83] Manolis G.H. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI," Report No. UCB/CSD 83/141, Computer Science Division (EECS), University of California at Berkeley, CA 94720 (October 1983).

[KEED78] J. L. Keedy, "On the Use of Stacks in the Evaluation of Expressions," *Computer Architecture News* **6**(6), pp.22-8 (February 1978).

[KEED78a] J. L. Keedy, "On the Evaluation of Expressions Using Accumulators, Stacks and Storage-to-Storage Instructions," *Computer Architecture News* **7**(4), pp.24-7 (December 1978).

[KEED79] J. L. Keedy, "More on the Use of Stacks in the Evaluation of Expressions," *Computer Architecture News* **7**(8), pp.18-22 (June 1979).

[KEED83] J. L. Keedy, "An Instruction Set for Evaluating Expressions," *IEEE Transactions on Computers* **C-32**(5), pp.476-8 (May 1983).

[KERN78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey 07632 (1978).

[KERN81] B. W. Kernighan, "Why PASCAL is Not My Favorite Programming Language," Internal Report, Bell Laboratories, Murray Hill, New Jersey 07974 (July 18, 1981).

[KRAL80] M. Kraley, R. Rettberg, P. Herman, R. Bressler, and A. Lake, "Design of a User-Microprogrammable Building Block," *Proc. of the 13th Annual Workshop on Microprogramming,* pp.106-14 (December 1980).

[LAMP82] B. Lampson, "Fast Procedure Calls," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.66-76 (March 1982). Published in *Computer Architecture News* **10**(2).

[LEVE83] Bruce W. Leverett, *Register Allocation in Optimizing Compilers,* UMI Research Press, Ann Arbor, Michigan (1983). (Ph. D. Dissertation).

[LEVY82] H. M. Levy and D. W. Clark, "On the Use of Benchmarks for Measuring System Performance," *Computer Architecture News* **10**(6), pp.5-8 (December 1982).

[LION79] John Lions, *The Second Pass of the Portable C Compiler,* Bell Laboratories, Murray Hill, New Jersey 07974 (June 1979).

[LUND77] Amund Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Communications of the ACM* **20**(3), pp.143-53 (March 1977).

[MACG84] D. MacGregor, D. Mothersole, and B. Moyer, "The Motorola MC68020," *IEEE Micro* **4**(4), pp.101-18 (August 1984).

[MC79] *MC68000: 16-bit Microprocessor User's Manual,* Motorola (1979).

[MCDA82] G. McDaniel, "An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.167-76 (March 1982). Published in *Computer Architecture News* **10**(2).

[MCKU84] Marshall K. McKusick, "Register Allocation and Data Conversion in Machine Independent Code Generators," Ph.D. Dissertation (Report No. UCB/CSD 84/214), Computer Science Division (EECS), University of California at Berkeley, CA 94720 (December 1984).

[MIRO82] James C. Miros, "A C Compiler for RISC I," Master's Project Report, Computer Science Division (EECS), University of California at Berkeley, CA 94720 (August 1982).

[MYER77] G. J. Myers, "The Case Against Stack-oriented Instruction Sets," *Computer Architecture News* **6**(3), pp.7-10 (August 1977).

[MYER78] G. J. Myers, "The Evaluation of Expressions in a Storage-to-Storage Architecture," *Computer Architecture News* **6**(9), pp.20-3 (June 1978).

[MYER82] Glenford J. Myers, *Advances in Computer Architecture,* John Wiley & Sons (1982).

[NCR83] *NCR/32 General Information,* NCR Corporation, Dayton, Ohio (1983).

[NORT83] R. L. Norton and J. A. Abraham, "Adaptive Interpretation as a Means of Exploiting Complex Instruction Sets," *The 10th Annual International Symposium on Computer Architecture*, pp.277-82 (June 1983). Published in *Computer Architecture News* **11**(3).

[OHRA84] Richard Ohran, "Lilith and Modula-2," *BYTE* **9**(8), pp.181-192 (August 1984).

[OLLE85] B. Ollerton, "Performance Architecture for the UNIX Environment," Internal Report, Celerity Computing Inc. (1985).

[OSBO81] Adam Osborne and Gerry Kane, *Osborne 16-bit Microprocessor Handbook,* OSBORNE/McGraw-Hill, Berkeley, Ca 94710 (1981).

[PATT82] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *Computer* **15**(9), pp.8-21 (September 1982).

[PATT82a] D. A. Patterson and R. S. Piepho, "RISC Assessment: A High-Level Language Experiment," *Proc. 9th Symposium on Computer Architecture*, pp.3-8 (April 1982). Published in *Computer Architecture News* **10**(3).

[PATT85] David A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM* **28**(1), pp.8-21 (January 1985).

[PEEK83] James B. Peek, "The VLSI Circuitry of RISC I," Report No. UCB/CSD 83/135, Computer Science Division, University of California, Berkeley, CA 94720 (August 1983).

[PIEP81] Richard S. Piepho, "Comparative Evaluation of the RISC I Architecture via the Computer Family Architecture Benchmarks," M. S. Project Report, University of California, Berkeley (August 27, 1981).

[POST83] Ed Post, "Real Programmers Don't Use PASCAL," *Usenet distribution* (1983).

[PRZY84] S. A. Przybylski, T. R. Gross, J. L. Hennessy, N. P. Jouppi, and C. Rowen, "Organization and VLSI Implementation of MIPS," Technical Report No. 84-259, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-2192 (April 1984).

[PTC83] "RISC Theory as Applied in the Pyramid 90x," Internal Report, Pyramid Technology Corporation (September 1983).

[RADI82] G. Radin, "The 801 Minicomputer," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.39-47 (March 1982). Published in *Computer Architecture News* **10**(2).

[SCHN71] Victor Schneider, "On the Number of Registers Needed to Evaluate Arithmetic Expressions," *BIT* **11**, pp.84-93 (1971).

[SCHU84] Peter U. Schulthess, "A Reduced High-Level-Language Instruction Set," *IEEE Micro* **4**(3) (June 1984).

[SEQU82] C. H. Sequin and D. A. Patterson, "Design and Implementation of RISC I," Report No. UCB/CSD 82/106, Computer Science Division, University of California, Berkeley, CA 94720 (October 1982).

[SETH70] R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Journal of the Association for Computing Machinery* **17**(4), pp.715-28 (October 1970).

[SHUS78] Leonard J. Shustek, "Analysis and Performance of Computer Instruction Sets," Ph. D. Dissertation (STAN-CS-78-658), Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94305 (January 1978).

[SITE79] Richard L. Sites, "How to Use 1000 Registers," *Caltech Conference on VLSI*, pp.527-32 (January 1979).

[SMIT82] A. J. Smith, "Cache Memories," *ACM Computing Surveys* **14**(3), pp.473-530 (September 1982).

[SMIT85] J. E. Smith and J. R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Transactions on Computers* **C-34**(3) (March 1985).

[SNOW81] E. A. Snow and D. P. Siewiorek, "Implementation and Performance Evaluation of Computer Families," *IEEE Transactions on Computers* **C-30**(6) (June 1981).

[STRE78] W. D. Strecker, "VAX-11/780—A Virtual Address Extension to the DEC PDP-11 Family," *AFIPS Conference Proceedings* **47**, pp.967-80 (June 1978).

[SWEE82] R. Sweet and J. G. Sandman, Jr., "Static Analysis of the Mesa Instruction Set," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.158-66 (March 1982). Published in *Computer Architecture News* **10**(2).

[TAMI83] Yuval Tamir and C. H. Sequin, "Strategies for Managing the Register File in RISC," *IEEE Transactions on Computers* **C-32**(11) (November 1983).

[TANE78] Andrew S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Communications of the ACM* **21**(3), pp.237-46 (March 1978).

[TANE83] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Communications of the ACM* **26**(9), pp.654-660 (September 1983).

[UNGA84] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," *The 11th Annual International Symposium on Computer Architecture*, pp.188-97 (June 1984). Published in *Computer Architecture News* **12**(3).

[WAKE83] Scott Wakefield, "Studies in Execution Architectures," Technical Report No. 237, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA. 94305 (January 1983).

[WEIC84] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM* **27**(10), pp.1013-30 (October 1984).

[WICH76] B. A. Wichmann, "Ackermann's Function: A Study in the Efficiency of Calling Procedures," *BIT* **16**(1), pp.103-10 (1976).

[WIEC82] C. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.177-84 (March 1982). Published in *Computer Architecture News* **10**(2).

[WULF81] W. A. Wulf, "Compilers and Computer Architecture," *Computer*, pp.41-7 (July 1981).

[YUEN84] C. K. Yuen, "Some Application of the Implicit Register Reference," *Computer Architecture News* **12**(1), pp.58-63 (March 1984).