A Stack-Based Register Set

Gordon Russell Paul Shaw

May 13, 1993

Department of Computer Science
University Of Strathclyde
26 Richmond Street
Glasgow
G1 1XH
Scotland

Abstract

Register windowing has become a common sight in high speed processors, reducing the memory traffic required to preserve register contents over subroutine invocations. However, approaches to register windowing have changed little since their introduction.

In this article, the current windowing schemes (namely fixed-sized and variable-sized) are presented, along with methods of implementation. A number of disadvantages with these systems are identified, and the requirements for an improved register windowing mechanism defined. A new register windowing paradigm is presented, known as *Shifting Register Windows*, which is designed to meet these requirements. This windowing system is first given in overview, followed by a more detailed description of the underlying model. Finally, a performance analysis of the model is presented, and conclusions drawn on speed and efficiency.

1 Introduction

The use of registers has grown considerably since the accumulators of von Neumann's 1945 machine. Index registers appeared in 1951 [5] as part of the Manchester University Digital Computing Machine, followed in 1956 by general-purpose registers (within the Pegasus computer from Ferranti) [8][page 10]. General-purpose registers are used to hold commonly accessed data, such as local variables, pointers, parameters, and return values. They cannot however, hold heap-based variables or other aliased data [4][page 116].

One problem with the use of general-purpose registers is in the overhead incurred over subroutine calls, where register contents must be saved to memory and restored on return. In [4][page 450], Hennessy and Patterson show that this overhead equates to between 5% and 40% of all data memory references. The common solution is to use many on-chip registers.

Large register files are managed either by software or hardware. In architectures where all general-purpose registers are viewed as a single register file, software techniques [7, 10] use global program knowledge to attempt to maintain values in registers over subroutine calls. To gain this global knowledge, register allocation is carried out at link time.

Hardware management strategies centre around *register windows*. Here, the register file is split into several banks, with a bank allocated on each call, and deallocated on return. The on-chip banks are constructed as a circular buffer: when a bank is requested that would result in a previously allocated bank being overwritten, the information contained therein is saved to memory (*window overflow*). On returning to a previously saved register bank, that bank is loaded from memory (*window underflow*).

Software techniques for maintaining values in registers have the advantage that the hardware is kept simple. However:

- Linking for a windowed register file is faster, and dynamic linking is easier to support.
- In the software solution, more directly addressable registers means more instruction bits are required to identify operands.
- Adding registers in a windowed architecture is transparent to the instruction set (and the user), while adding to a non-windowed system is not.

It should be stated that register windows cannot readily replace *all* processor registers, since globally accessible registers will still be required (*e.g.* program counter, user stack pointer, window overflow stack pointer, *etc.*). Although windowing floating-point registers is possible, current architectures typically leave these registers global.

Register windowing can be split into two general sub-classes: fixed- and variable-sized. In a fixed-sized register windowing scheme, the number of registers per bank is defined by the hardware designer, whereas in a variable-sized scheme, the bank's size is specified by software at allocation time. These two systems are discussed in the following sections.

1.1 Fixed-Sized Register Windows

Fixed-sized register windows are used by both the SPARC chip-set [3] and the RISC II [6]. For these processors, the active window (*i.e.* currently accessible block of registers) is split into three parts: *in*, *local*, and *out*, with each holding eight registers. The *local* part contains registers accessible only while that window is active, *out* holds parameters to be passed to subroutines, and *in* holds the current subroutine's parameters as supplied by the parent. Whenever a new window is created, the *out* registers of the previous window become the *in* registers of the new window. This mapping is undone when the new window is deallocated.

Three fixed-sized windows are shown in figure 1. Each column represents the parts accessible from any one window. Parts lying on the same row are directly mapped onto one another. For example, the *out* part of window 2 is mapped directly onto the *in* part of window 3. The underlying register file is shown on the right.

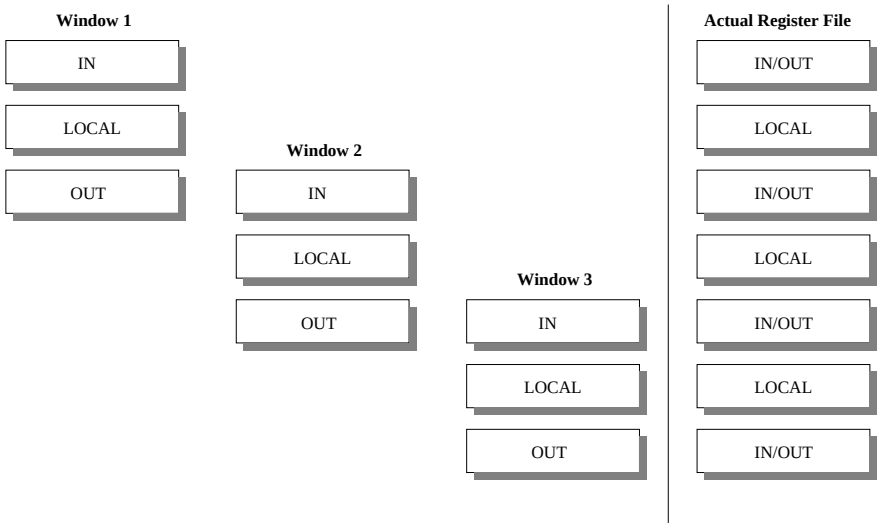

Figure 1: Three Fixed-Sized Register Window Banks

Increasing the register file size increases internal bus capacitance. Provided that number of windows is kept small, this does not appear to effect processor cycle time [4][page 454]. However, with many on-chip banks, cycle time will certainly be affected, suggesting an upper limit on design scalability.

Fixed-sized windows offer no flexibility in the number of parameters passed or locals declared. If the number of parameters exceed the size of the *in* register part, then the remaining parameters must be held in memory. Alternatively, if some registers within a bank are unused within a subroutine, then window overflow/underflow will involve redundant memory transfers.

1.2 Variable Register Windows

An organization supporting variable-sized register banks is shown in figure 2. Here, a global register stores the current window position. Its value is added to every register reference, and then passed to a decoder, selecting the desired register.

The only instruction used in controlling the windows is a *shift*. On a subroutine call, the parent shifts the current window position to select the first parameter to be passed (*i.e.* a position after the parent's local variables). A negative shift undoes this on return from the subroutine. Once called, a subroutine can access registers from the current window pointer onwards. The Am29000 [1] provides support for variable-sized windows in a similar way to that depicted.

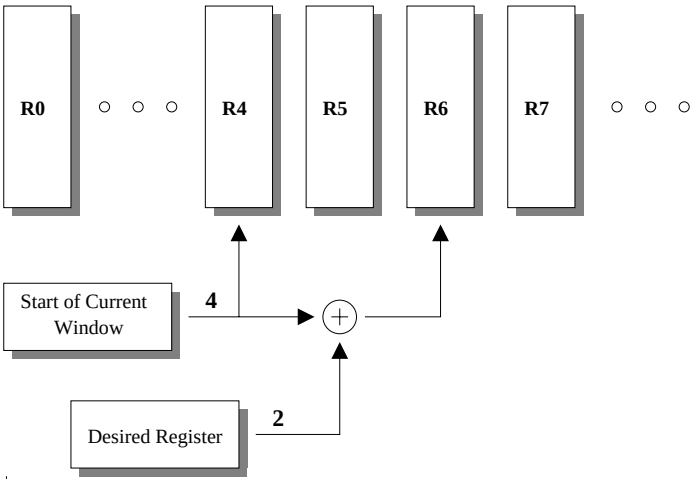

Figure 2: Organization of Variable-Sized Register Windows

Although this scheme supports variable-sized windows, there is an overhead of an addition on each register access. The problem of scalability identified with fixed-sized windows remains unsolved.

1.3 Design Goals for a Better Windowing Mechanism

Each of the existing register management systems described demonstrates advantages over the other. For example, variable-sized windows offer flexibility at the cost of performance, while the fixed scheme provides better performance at the expense of flexibility.

A new windowing model has been constructed; *shifting register windows*. Its aim is to improve over existing register management schemes, such that:

- The new design contains the flexibility of variable-sized register windows.
- Currently accessible registers (those contained within the active window) should be the registers closest to the ALU, minimizing signal propagation times.
- The design should be without register-access overhead, such as that incurred by the addition in variable-sized windows.
- The return address for a subroutine should be stored on-chip to support fast call-return cycles.

- The design should be scalable, such that adding more on-chip space for register storage does not adversely effect access times or logical complexity.

2 Shifting Register Windows

Multiple accesses to the same register within a variable-sized windowing scheme each require an addition, even if the current window position remains unchanged between accesses. This can be avoided by keeping the active window pointer fixed and moving the contents of the register file instead. A shifter can be used to obtain this functionality.

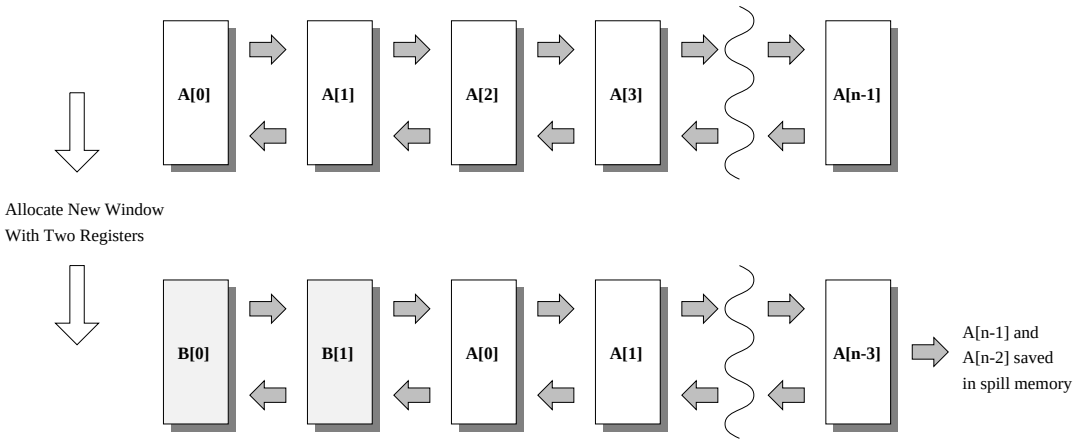

Figure 3: Register File Arranged as a Shifter

Figure 3 shows a register file $A[0 \ldots (n-1)]$ arranged as an n cell shifter. It also shows the allocation of a new window B, containing two elements. This window was allocated by performing two right shifts, and will later be deallocated by two left shifts (and its contents lost). After allocation, registers $B[0,1]$ occupy the *same physical cells* as $A[0,1]$ did before allocation. Therefore, the active window is always located in the leftmost cells. Processor designers should arrange for this area to be as close the the ALU as possible.

When registers are allocated by shifting, information contained within the rightmost register cells will be lost. To avoid this, register contents shifted off should be stored in memory. In figure 3, registers $A[(n-2),(n-1)]$ have been saved. On deallocation, register contents residing in memory should be returned to the register file. This implies that memory accesses are performed in step with shifting, requiring the processor to stall until the accesses have completed.

This stalling problem can be alleviated by allowing the shifter to expand and contract as necessary. In this way, some elements which would have been spilled can be absorbed within shifter's elasticity. Whenever the shifter has been enlarged in this way, a secondary system migrates those elements which caused the growth into memory. A similar mechanism operates when the shifter is under-full[1]. The migration process stops when the shifter's capacity returns to normal, or in the case of contraction, there are no more elements stored in memory.

An elastic shifter is constructed from n elastic cells. To the left of each cell lies a *shadow* cell. Both cells are capable of holding one register element. Left or right shifting requests are injected into the leftmost cell of the shifter, and propagate between elastic cells. Acknowledgements are propagated right to left in response to requests. During a shift, the shadow cells are used as intermediate storage. Requests persist until acknowledged or cancelled, and at any time both left and right shift requests may exist along the length of the shifter (although only one type of request may exist between any two

[1] This can occur when elements have been moved to memory and then registers deallocated.

neighbouring elastic cells). Depending on the type of requests, the shifter can appear to contain from 0 (a left shift persists for every elastic cell in the shifter) to $2n$ registers (a right shift persists for every elastic cell in the shifter). When the shifter contains no outstanding requests, then it is said to be *stable*.

The fact that requests must propagate from cell to cell means that it takes longer to stabilize than the fixed-length variety. Without global knowledge of persisting requests, a shifter must stabilize before the location of a register can be predicted.

2.1 Overview

Figure 4 shows the organization of shifting register windows within a processor. The diagram depicts five main entities: the interface from the registers to the control unit, the *active window*, *passive window*, *spill manager*, and *spill memory*.

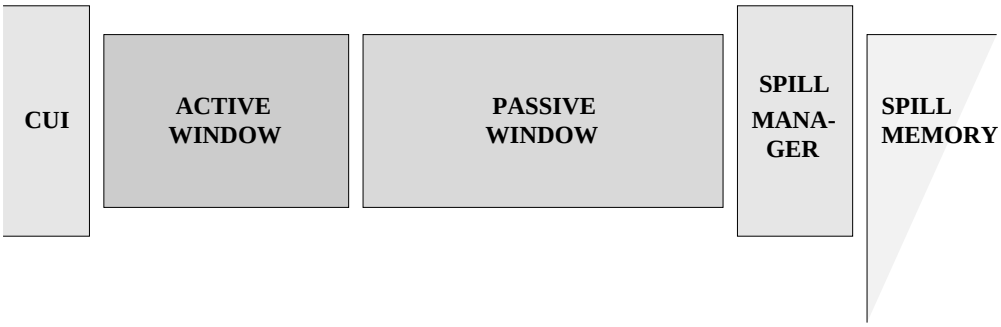

Figure 4: The Register Windowing System

- The **control unit interface (CUI)** allows the control unit (CU) to interface with the register file. This interface takes register allocation/deallocation instructions from the CU and converts these to simple shifting commands. These commands are then transferred to the active and passive windows.

- The **active window** is a synchronous shifter comprised of a user-accessible set of registers (accessed by busses which span its length). Its synchronous nature allows it to stabilize quickly.

- The **passive window** forms a elastic shifter of non-user-accessible cells. It receives shifting commands from the CUI and buffers data transfers between the active window and spill manager. Since the processor never accesses the passive window directly, register busses are limited to those cells in the active window.

- The **spill manager** responds to requests generated by the rightmost cell of the passive window. When requested to shift right, the spill manager will transfer the data to be shifted into memory. The opposite transfer occurs on a left shift. Spilled registers are stored in a last in first out manner. In an attempt to minimize processor stalls, the spill manager's memory transfers are undertaken only when free memory cycles are available. It is recommended that an instruction cache is included in the processor, promoting these free memory cycles.

- **Spill memory** holds register contents transferred by the spill manager during right shifts. Depending on decisions taken during the processor design stage, accesses to spill memory can be normal cached accesses, or go directly to main memory.

2.2 The Complete Shifter

In shifting register windows, the active window is composed of a synchronous cells. The passive window contains b elastic cells, with a shadow cell to the left of each (thus having a maximum capacity of $2b$ data items). Data can be shifted between the last element of the active window and the first shadow cell of the passive window.

During a right shift, the CUI instructs the active window to shift to the right. Once the active window has completed shifting, the data shifted off the end of the active window resides in the first shadow cell of the passive window. A right shift request is then sent to the passive window from the CUI. When successfully acknowledged, new commands can be delivered to the active window. For a left shift, first the passive window is requested to shift left, and when positively acknowledged the active window is also shifted left.

In both shifting scenarios, there is no requirement for the passive window to stabilize before the CUI sends it subsequent shift requests. Previous requests propagate along the passive window independently from current requests.

2.3 A Single Register Cell

There are three types of cell used in constructing the overall model, known as *synchronous*, *elastic* and *shadow*. All hold the same type of information:

- A data part, which holds either a program-related variable or a subroutine return address.
- A valid bit, indicating whether the data part is currently in use.

2.3.1 Synchronous Cell

Two global control lines are connected to each synchronous register cell. (SHIFT-R) signals a cell to shift out its contents to its right-hand neighbour, and shift in the contents of its left-hand neighbour. (SHIFT-L) signals the inverse operation.

2.3.2 Elastic and Shadow Cells

The elastic cell is more complex, using handshake lines to communicate shifting actions between itself and neighbouring cells in the shifter. Each pair of neighbouring elastic cells share an intervening shadow cell, through which data to be shifted is communicated. This organization is shown in figure 5. This use of shadow cells is similar to that presented in [2]. In the steady state, all data is held within elastic cells, with the shadow cells remaining empty.

Figure 5 shows two asynchronous handshake channels between each pair of elastic cells. Each of these channels is made up of three wires: a request line REQ and two acknowledgement lines OK and FAIL as shown in figure 6. All requests are made from the left. On any one channel, a request must be acknowledged before a subsequent request can be issued. A request is successful if OK is acknowledged and unsuccessful if FAIL is acknowledged. Channel ESHIFT-R (elastic shift right) controls right shifts, and ESHIFT-L controls those to the left. A request to a cell fails if that cell is currently issuing the same class of request to its right hand neighbour.

The operation of a pair of elastic cells during ESHIFT-R and ESHIFT-L requests is as follows:

- An ESHIFT-R request made from C_i to C_{i+1} instructs C_{i+1} to first move its data to its right shadow cell, then move the contents of its left shadow cell into itself. The request is successful if this could be done, or unsuccessful otherwise.
- An ESHIFT-L request made from C_i to C_{i+1} instructs C_{i+1} to move its data into its left shadow cell. If this could be done, the request is successful, and C_i then moves data from its right shadow cell into itself, otherwise the request fails.

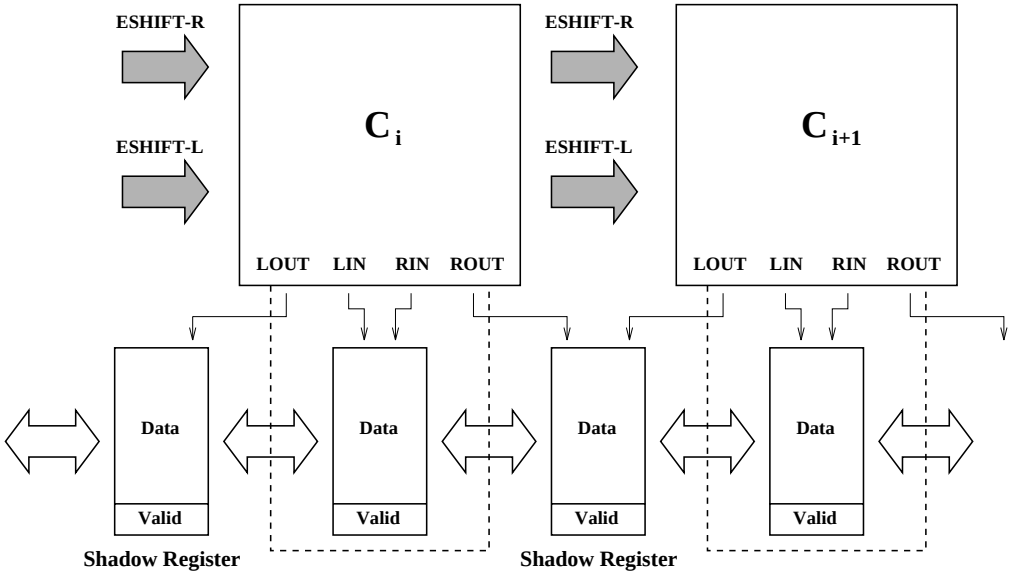

Figure 5: An Elastic Register Cell and its Neighbours

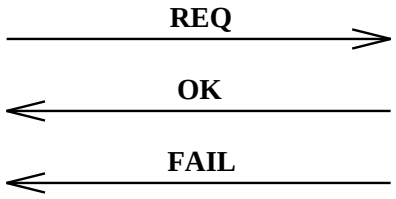

Figure 6: Composition of a Handshake Channel

The movement of data between shadow and elastic cells is controlled by four signals: ROUT, RIN, LOUT, and LIN. ROUT latches data from the signalling elastic cell to its right hand shadow cell, while RIN latches in the opposite direction. LOUT and LIN control the symmetric operations for the left hand shadow cell.

Each elastic cell contains a state variable *status* which takes on one of three values: IDLE, DO-ESHIFT-R or DO-ESHIFT-L. This defines what the cell should be doing when receiving no requests from its left neighbour. The *state* of a cell is defined to be the value of its *status* variable.

In the IDLE state C_i does nothing. In the DO-ESHIFT-R state, C_i makes an ESHIFT-R request to C_{i+1}. If successful, C_i assumes an IDLE state. Otherwise, its state remains unchanged, and the ESHIFT-R request will be tried again some time later. A similar action is done in the DO-ESHIFT-L state. If previously in the IDLE state, C_i enters state DO-ESHIFT-R when an ESHIFT-R request from C_{i-1} succeeds. Similarly, C_i enters state DO-ESHIFT-L when a ESHIFT-L request from C_{i-1} succeeds.

Requests issued by C_i fail when the state of C_{i+1} matches the type of request. That is, if C_{i+1} is in state DO-ESHIFT-R and receives an ESHIFT-R request from C_i, the request fails, and the state of C_i remains unchanged. Likewise, ESHIFT-L requests issued by C_i fail when C_{i+1} is in state DO-ESHIFT-L.

If the state of C_{i+1} does no match the request from C_i, optimizations occur. For instance, if C_{i+1} is in state DO-ESHIFT-R and an ESHIFT-L request is received from C_i, the request is successful, the previous register transfers are undone, and C_{i+1} assumes the IDLE state. Similar actions occur if C_{i+1} is in state DO-ESHIFT-L and an ESHIFT-R request is received from C_i.

2.4 The Spill Manager

The spill manager is an augmented elastic cell. The interface to its left cell remains unchanged, with a shadow cell present between itself and its neighbour. The cell's right-hand interface is not to another elastic cell, but to spill memory.

The manager can exist in one of three states; IDLE, DO-ESHIFT-R, and DO-ESHIFT-L. During the IDLE state, the manager waits for requests from its left hand neighbour.

When in the IDLE state, reception of an ESHIFT-R request causes the valid bit of the spill manager's internal data to be examined. If the data is invalid, the request succeeds, the state remains unchanged, and the transfer from shadow cell to spill manager performed. If the data is valid, the request fails, and the spill manager assumes state DO-ESHIFT-R.

ESHIFT-L requests received in the IDLE state are successful if the data valid bit is set or the spill memory is empty[2]. Successful ESHIFT-L requests result in the normal data transfer from manager to shadow cell, and invalidation of the manager's valid bit. Whether the request is successful or not, the state is set to DO-ESHIFT-L if the spill memory is non-empty.

State DO-ESHIFT-L indicates that a data item previously saved in the spill memory should be reloaded (and the valid bit set). DO-ESHIFT-R indicates the opposite action; data should be moved from the manager to spill memory (and the valid bit cleared). The memory transfers are delayed until both a free memory bus cycle is available and a hysteresis condition is met. When spilling, the hysteresis condition is that the last h elastic cells of the passive window are in state DO-ESHIFT-R. Refilling requires these h cells to be in state DO-ESHIFT-L. The hysteresis reduces thrashing (continual loading and storing of the same data elements). When the memory transfer completes, the spill manager returns to the IDLE state. During spill-memory transfers, requests are unsuccessful.

Whenever the spill manager's state matches the input request, that request will fail. With the state and request opposing, requests succeed provided the spill-memory transfer has not started. Receiving ESHIFT-L in state DO-ESHIFT-R places the manager in state DO-ESHIFT-L, and allows the manager to shadow cell transfer to complete normally. Receiving ESHIFT-R in DO-ESHIFT-L sets the state to IDLE, and initiates the required shadow to manager transfer.

If the spill manager is starved of free bus cycles, then it is possible for the passive window to become either completely full or completely empty. In both cases, the processor cycles waiting for a

[2] The amount of information stored in spill memory could be held within a counter.

positive acknowledgement from the passive window. This results in an increased number of free bus cycles, allowing the spill manager to make the necessary memory transfers. Transfers made while the processor is stalled are referred to as *forced* memory transfers. On a traditional windowed machine (*e.g.* SPARC), all memory transfers to handle window spills and refill are forced: the processor can do no useful work during this time.

Typically, when the passive window contains $b - h$ data items or less, the spill manager attempts to pull data from memory. Likewise, when it contains $b + h$ items or more, the spill manager attempts to store data.

2.5 Register Interfacing

This proposed interface to shifting register windows (ignoring supervisory instructions) consists of four instructions. These are ALLOC, DEALLOC, CALL and RETURN. The ALLOC and DEALLOC instructions allow the machine to acquire and relinquish registers, while the CALL and RETURN instructions use the register windows to store and restore the program counter. It should be possible to allocate and deallocate several registers per CPU cycle. If the number of registers which can be allocated/deallocated per cycle exceeds the number of registers requested, then ALLOC and DEALLOC will complete within a single cycle (otherwise these instructions take two or more cycles to complete). The operation of each of these instructions is described in program 1.

Three primitives are referenced, namely IN, OUT and SIGNAL. The IN and OUT operations correspond to right and left shifts, and SIGNAL corresponds to a hardware trap. reg is the register file of dimension equal to the size of the visible window, addressed from 0 upwards. IN takes two parameters: the data to be inserted and the valid bit state. OUT requires only one parameter: where to put the data element produced by the shift left (which in this case is either to store it in the PC, or to throw it away).

2.6 Startup and Context Switching

In a multitasking system, the kernel handles process loading, saving, and initialization. All process registers and selected internal state variables must therefore be readable and writable. The passive window does, however, complicate the kernel, since no data items within this window are directly accessible to the processor. Instead, the contents of the active and passive window must be flushed to spill memory on a context save, and restored again when the task is reloaded.

Flushing the registers to memory is accomplished by turning off hysteresis, and performing $a+b+1$ right shifts. When all elastic cells and the spill manager are in the IDLE state (as indicated by a global line[3]), the flushing is complete. Data elements saved during a previously context switch are selected by setting the internal variable which points to the spill area. The new context's data elements are then loaded into the register file by performing $a+b+1$ left shifts. Hysteresis is then switched back on. The reloaded process resumes execution as soon as a elements have been reloaded and propagated along the passive window into the active window. Any remaining transfers will continue to occur on free bus cycles under the control of the spill manager.

If a process is beginning for the first time, the register file is initialized by setting the spill count to zero. Global lines can then be used to invalidate all valid bits, and set the states of all elastic cells (and the spill manager) to IDLE.

With a shifting register windowing implementation, the overhead of context switching depends on the number of register saves and restores made by the spill manager during the switch. This overhead is no larger than that of a similarly sized (in terms of the total number of register holding elements) traditional fixed- or variable-sized scheme.

[3] In order to allow this global line sufficient time to stabilize, is should be possible to suspend the operation of all elastic cells (*e.g.* in a synchronous implementation, by removing their clocks at source). The stopping and starting of elastic cells is only performed during a context switch.

```
ALLOC(num)
    FOR i := 1..num
        IN(0,TRUE)

DEALLOC(num)
    FOR i := 1..num
        IF (reg[0].valid)
            OUT(null)
        ELSE
            SIGNAL(STACK_EMPTY)

CALL(location)
    IN(PC,TRUE)
    JUMP location

RETURN(num)
    DEALLOC(num)
    IF (reg[0].valid)
        OUT(PC)
    ELSE
        SIGNAL(STACK_EMPTY)
```

Program 1: User-Level Interface

2.7 Implementation

An implementation of shifting register windows has been designed by the authors. This particular implementation is synchronous, allowing simple interfacing to a synchronous processor. Around 50 gates are required in each elastic cell to control handshaking. Analysis suggests allocation/deallocation of a register occurs in approximately 15 gate times.

3 Performance

The main performance benefit of shifting register windows comes from the improved access speed of the register contents, due to short bus lengths. With an 128 element register file, transferring to a shifting register window implementation with a active window of 16 elements could result in an eight-fold decrease in bus length. Predicting the performance impact of short busses is dependent on a large number of implementation-specific factors. However, the amount of spill-memory accesses and register management related processor stalls is implementation independent, and can be modelled unambiguously.

To predict memory accesses caused by spilling and restoring registers and processor stalls introduced due to register management, a simulator was constructed. This traced SPARC binaries using SHADOW[9], and modelled instruction and data caches, SPARC fixed-sized windows, and shifting register windows. This allowed comparison of shifting register windows to be made against the SPARC windowing system, and demonstrated the effect that shifting register windows has on memory bus contention. Note the results are for one process running to completion, with no context switches.

In the simulation, the instruction cache was 8 KBytes and the data cache 4 KBytes. Both caches were direct mapped, with a block size of 16 bytes. The data cache was write back. The SPARC was

assumed to have seven windows, with a spill/refill of one window costing 60 cycles [4][page 452].

For the shifting register windows simulation, an active window size of 16 cells was used. The spill manager had a hysteresis of eight elastic cells, and bypassed the data cache when accessing spill memory. It was assumed that a spill/refill of a single register required four cycles, and that six registers could be allocated per CPU cycle.

In order to reduce contention between cache-miss induced memory accesses and spill-memory accesses, a pipeline *look-ahead* was used. This warns the spill manager of forthcoming data cache accesses using information gleaned from the pipeline. Such a warning prevents the spill manager from initiating spill memory accesses. In the simulation, a look-ahead of two cycles was assumed.

The benchmarks used were TEX and Gcc (used throughout [4]), DeTEX, zoo, and fig2dev. The passive window size was varied from 16 to 64 elastic cells. Two graphs were produced, detailing ratios of shifting register windows against SPARC windows for spill/refill memory accesses (figure 7) and processor stalls (figure 8). Program stalls include memory collisions between spill manager and caches, spills when the passive window is completely full, and refills when the passive window is completely empty. Associated with the refill is the time taken for the last register loaded to propagate to the active window. In practice, this propagation time was found to be negligible.

From figure 7, shifting register windows appears to generate less memory traffic than the SPARC windowing system for passive window sizes over approximately 24. Maintaining low memory traffic is a goal in multi-master systems.

Figure 8 shows that typically, processor stalls for shifting register windows are less than that incurred by the SPARC, for passive window sizes over 16. By reducing processor stalls, instruction throughput is increased.

The hysteresis value chosen in the simulation was a balance between two performance-related factors: forced spill-memory accesses and cache collisions. A low hysteresis value results in fewer forced memory accesses at the expense of increased cache collisions. The converse situation is true for a high hysteresis value. In terms of circuit area and signal propagation times, a low hysteresis value is desirable. In a processor design, the exact figure chosen will also depend on cache miss rates, look-ahead distance, and the speed of external memory.

4 Conclusions

The proposed shifting register windowing mechanism offers substantial improvements over existing schemes. The main benefits are minimization of register bus length, a reduction in spill/fill overhead, and automatic zeroing of local registers.

These improvements come at the cost of control circuitry and increased time taken allocate local registers: allocation time is proportional to the number of registers requested. Processor designers should strive to achieve sufficient allocation rates such that the majority of register allocations can be handled in a single cycle. Since the passive window is the slowest part of the design, allocation rates can be improved by using two passive windows, with the active window communicating to each in turn. The performance gain comes at the expense of a more complex, two port spill manager.

It is hoped that the ideas presented in this article will both increase the performance of register-based processors, and encourage further research into register windowing paradigms.

Acknowledgement

The authors would like to thank Paul Cockshott and Robert Lambert for providing useful feedback on earlier drafts of this article.

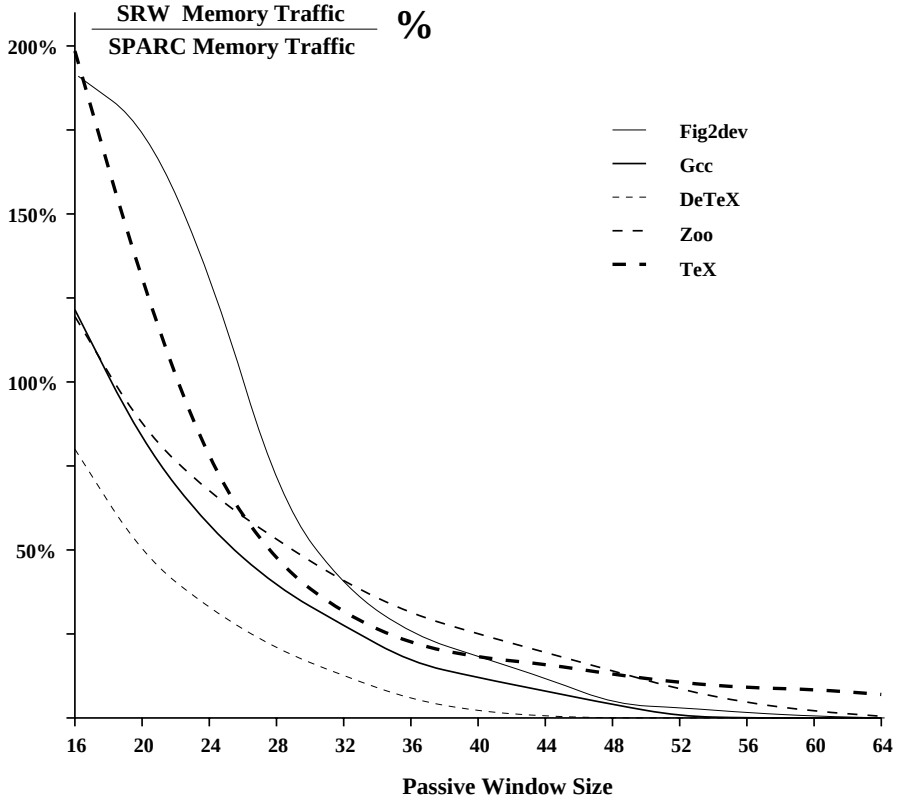

Figure 7: Shifting Register Window's Memory Accesses

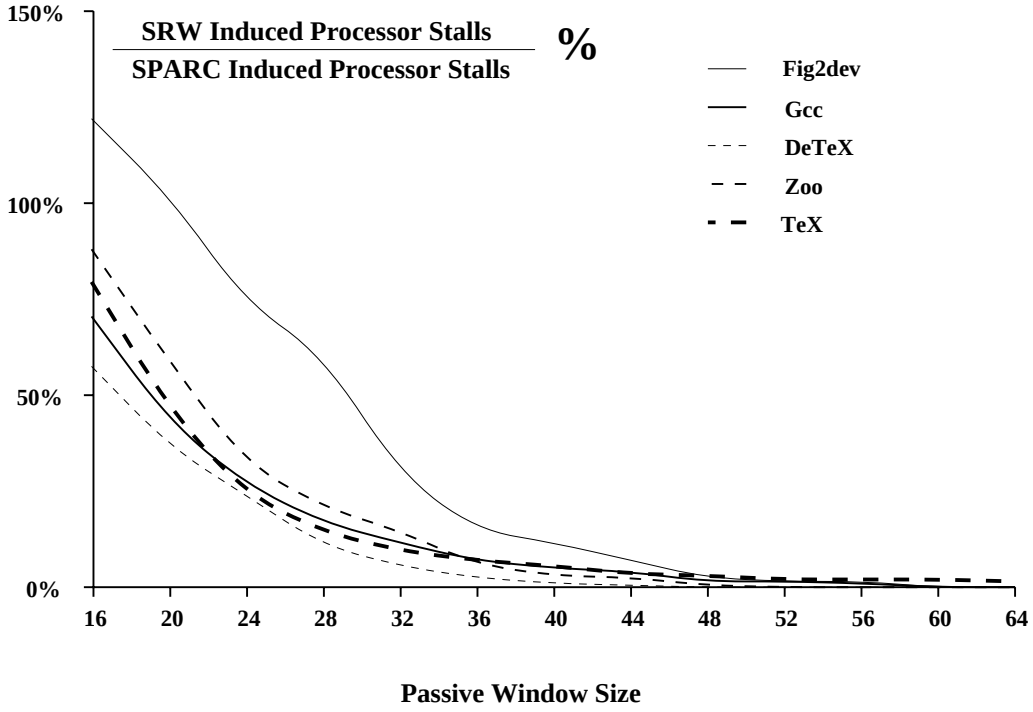

Figure 8: Shifting Register Window's Processor Stall Cycles

References

[1] Advanced Micro Devices, 901 Thompson Place, PO Box 3453, Sunnyvale, California 94088. *Am29000 User's Manual*, 1987.

[2] Jo C. Ebergen and Sylvain Gingras. An asynchronous stack with constant response time. Technical Report CS-93-11, Computer Science Department, University of Waterloo, August 1992.

[3] Brett Glass. SPARC revealed. *Byte*, 16(4):295–302, April 1991.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1990.

[5] Tom Kilburn. The manchester university digital computing machine. In M. R. WIlliams and Martin Campbell-Kelly, editors, *Charles Babbage Institute Reprint Series for the History of Computing*, volume 14, pages 138–141. MIT Press, 1989.

[6] David A. Patterson and Carlo H. Séquin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, 1982.

[7] Stephen Richardson and Mahadevan Ganapathi. Code optimization across procedures. *Computer*, 22(2), February 1989.

[8] Daniel P. Seiwiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill International, 1982.

[9] SUN Microsystems. *Introduction to SHADOW*, April 1992.

[10] David W. Wall. Register windows vs. register allocation. Technical report, Digital Research, December 1987. Also published in *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.