Register Window Architecture for Multitasking Applications

D. Quammen
Department of Computer Sciences
George Mason University
Fairfax, VA 22030
quammen@gmuvax2.gmu.edu

D. R. Miller
MITRE Corporation
7525 Colshire Drive
McLean, VA 22102
rmiller@gmuvax2.gmu.edu

D. Tabak
Electrical and Computer Engineering Department
George Mason University
Fairfax, Va, 22030
dtabak@gmuvax.gmu.edu

Abstract

The organization of large register banks into windows has been shown to be effective in enhancing the performance of sequential programs. One drawback of such an organization, which is of of minor importance to sequential languages, is the overhead encountered when the register bank must be replaced during a task switch. With concurrent language paradigms, such as are found in Ada[1] , Occam, and Modula-2, these switches will be more frequent. We introduce here a methodology, and an architecture, which greatly reduces this overhead while maintaining the inherent advantages of the register window approach. In addition, we present ways of implementing traditional stacks and queues, as well as hierarchical storage structures using windows.

1 Introduction

Current VLSI technology can provide a relatively large bank of on-chip storage which is accessible at a much higher speed than off-chip memory. Providing on-chip room for this large register bank has been one of the motivating factors in reducing instruction (and therefore control unit) complexity [11, 15]. Use of these memory banks has yielded impressive performance gains for sequential languages [3, 17]. However, programs written using new language paradigms, such as object oriented languages and concurrent languages, cannot benefit from the same techniques used by sequential programs. One unfortunate side-effect of current on-chip memory organizations is that they need to be saved and restored whenever an environmental context switch occurs. This is particularly true in multitasking environments, and is distressing when considering program designs which feature many and frequent task context switches.

In this paper, we present a method to manage register windows which facilitates multitasking on a single register window bank. We also describe the architectural features to support this technique. To demonstrate the utility of this design, an outline for an implementation of the Ada tasking requirements is presented.

We are concerned, for the time being, only with the storage of data, rather than code. Data is dynamically created, destroyed, and altered. Its access characteristics tend to be temporal in nature, and its life time tends to be either very short, or very long. The run-time overhead needed to manage the dynamic ebb and flow of data is expensive. It is this specific area of a multitasking environment that we are addressing.

1.1 Organization of Large On-Chip Memory

A number of different approaches have been developed for organizing on-chip memory. Three of the more widely accepted approaches are a cache, a large directly addressable register bank, and a register (or register window) stack. There has been much discussion as to which of these methods is most appropriate [17, 8, 6, 5].

The first option, cache, has four major advantages;

1. control of cache loading and spilling is invisible to the programmer
2. cache addressing is identical to memory addressing and requires no additional compiler overhead
3. cache is responsive to dynamic behavior
4. cache can service both retentive storage, such as globals, and dynamically created storage.

These advantages have corresponding disadvantages. Since control is invisible to the user, the user cannot benefit from known or deduced program behavior. In addition since cache is mapped into memory when dynamic storage is created, memory as well as cache must be allocated. This double allocation is expensive, and is an undue overhead if the allocated memory is short-lived. Cache also requires more hardware per memory cell and tends to be slower than registers.

Many systems [10] utilize the second approach to managing large numbers of registers, i.e. they provide a large bank of general purpose registers. Registers can

[1] Ada is a registered trademark of the U.S. Government(Ada Joint Program Office).

use knowledge about the needs of a program to preload and purge data more efficiently [8]. Users of these systems have successfully avoided much of the overhead of copying registers at the point of a subroutine call by mapping register usage at compile time through either static or trace-generated call graphs [17]. These systems benefit by allowing frequently used variables, be they procedure locals or globals, to reside in registers. However, compiler processing is expensive, and it is not always possible to map every variable to a unique register. Such variables must either reside permanently in memory or be copied in and out of different registers.

The third technique uses the on-chip storage to hold the top of the procedure activation stack. This has been the work of Ditzel [4] and Patterson [11] This technique is based on the observed behavior of C and Pascal programs which concentrate most data reference activity to the storage at the top of the activation record stack. This storage is generally shortlived and is frequently deallocated (the procedure returns) before it overflows into memory. If there is an overflow, the allocation can be spilt into an easily managed memory resident stack.

There are several additional benefits to the register stack technique. One, since addressing can be done relative to the top of stack or within an allocated window, the size of the register address is small, and (to some degree) independent of the on-chip memory size. Two, compiler complexity is reduced since each subprogram can assume that it has dedicated registers. However, there are also disadvantages, recent studies [17, 6, 5] have indicated that the housing of frequently accessed globals may (in some situations) be a more beneficial use of registers than procedural activation records. In general, register stacks do not support this activity, although some do provide a limited number of global registers which can be used for this purpose. In addition, if the on-chip memory is large, much of the register bank could remain unused if procedure call chains did not achieve significant depth [16].

We here present a fourth arrangement which not only exhibits the benefits of the above mentioned systems but also lends itself to the implementation of additional data structures required in systems with non-LIFO dynamic storage allocation. None of the above systems address this domain.

1.2 Threaded Register Windows

Our goal is to design a system which offers the advantages of general registers, register stacks, and cache, but which minimizes some of the problems associated with them. We call the concept *Threaded Register Windows* [13]. Currently the basic unit of the thread is a register window, which contains 16 32-bit registers. The size of this window is experimental, and may be changed later. These windows do not overlap as do those of the Berkeley RISC-II, but dual pointers supported by the architecture allow access to both a calling procedure's activation window and the called activation window - therefore any passed parameters need not be copied. The windows can be flexibly and dynamically configured to serve as:

1. Activation Record Stacks: several windows may be dynamically linked together to form a stack to hold procedural activation records. Because no window overlap is required, the windows forming the stack need not be contiguous.

2. General Purpose Traditional Data Stacks: the same linking approach can be used to form a traditional pushpop data stack. Implicit access to this stack is supported by the architecture.

3. General Purpose Queues: the architecture also supports implicit access to queues made up of linked register windows.

4. Statically allocated windows to hold vital information for an interrupt handler or the operating system.

5. Isolated packets of frequently used global or object oriented data.

The structures are created using doubly linked lists. The overhead involved in forming the links is low, and the architecture supports constructs which make it transparent to application programs.

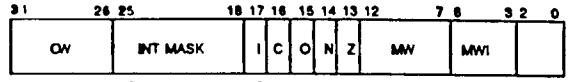
There are many advantages to these structures. The activation record stack allows the run time system to have multiple stacks (or at least the tops of them) resident on-chip at one time. This allows tasks to save their most recently used and needed data in the fastest storage available, and not copy it at the time of a context switch. The result of this makes the performance gains achieved by other RISC machines [3] available to multi-tasking systems.

The queue structure allows task communications to occur within the domains of the fast on-chip storage, and also facilitates scheduling. The additional availability of traditional stacks is advantageous since this structure is sharable by all procedures of a task or system. That is, it is not tied directly to the procedure control structures. This structure would be useful to a compiler; to hold data from large procedural activation records (those which require more than 16 words); or to support recursive algorithms, such as those used to handle trees and graphs [2].

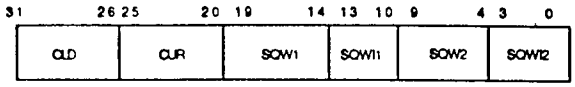
The interior of the windows is accessed using a block relative address (register number). Therefore, the operand address is short, allowing for three operand instructions, and the compiler can assume that a dedicated set (one window) of registers is available for local and parameter storage. Dynamically allocated storage is initially allocated only in the windows. This is done by taking a *free window* from a hardware supported

(chip resident) *free window stack*. Off-chip memory is allocated only if an overflow occurs. Which windows to overflow, or to underflow, can be computed in the background using information available from the operating system and guidance from the compiler. The additional processing time needed to choose which window to overflow can be regained by avoiding *misses* and unnecessary memory allocations.

This work has characteristics of several other machines, but none use identical techniques. The idea of register windows was first developed by Patterson [11]. However, his work is not efficiently extendible to non-sequential situations. The AM29000 [1] can extend register management to aid multitasking. The 128 registers can be configured as eight segments, each with 16 registers. However, these are of fixed size and are statically allocated, one per task. This can reduce procedure call/return performance. The concept of pointers to dynamic blocks was used in the Intel 432 [12]. However, this machine used many levels of indirection and did not exploit on-chip storage [7]. The IBM 801 [14] added additional cache commands to avoid some of the unnecessary overhead involved with cache (such as loading the contents of a newly allocated procedural activation record even though this data is void), however their techniques did not go far beyond this.



Program Status Word 1



Program Status Word 2

CUR	Current activation record window pointer
OLD	Previous (caller's) activation record window pointer
SQW1	Stack/Queue Window 1 pointer
SQWI1	Stack/Queue Window 1 index
SQW2	Stack/Queue Window 2 pointer
SQWI2	Stack/Queue Window 2 index
OW	"Object" Window pointer
MW	Map Window pointer
MWI	Map Window Index
ICONZ	Interrupt enable, carry, overflow, negative, and zero flags

Figure 1 - Processor State Registers

2 Register Window Thread Organization

In the register window thread organization register access is implicit through window pointers contained in the processor state. The processor state, saved in two process status words (PSW1 and PSW2), includes references to six windows at any one time. Additionally, there is always an implicit reference to the global window, giving access to a total of 7*16 registers. Access is restricted to only these windows. This allows the scope of a procedure to be inherently protected. Because of the limited address field, it is impossible to accidentally access data outside the windows which have been allocated to a process.

Figure 1 shows the organization of the processor state registers. Of the six windows, five are used to access user data. One, the *Map Window* (*MW* in PSW1) is used to manage activation record stacks. The five data window references are:

1. The current window (accessible through *CUR* in PSW2) - containing the activation record of the currently active procedure.

2. The old window (accessible through *OLD* in PSW2) - containing the activation record of the current procedure's dynamic parent. Any parameters which are passed, are accessed directly using this pointer.

3. StackQueue Window 1 and 2 (accessible through *SQW1* or *SQW2* in PSW2) - These windows may be treated as data stacks, the head or tail of a queue, or a general data windows.

4. An object window, (accessible through *OW* in PSW1) - this window can be used to hold any additional program-controlled data pointer which, for example, can be used to access an object or frequently referred to globals.

The *SQW* pointers also have an intra-register displacement associated with them. A register address is expressed in the instruction using two fields. The first is a three bit field indicating one of the seven accessible windows; the second a four bit field indicating one of the sixteen registers in that window.

The register window pointers (*CUR, OLD, SQW1, SQW2, OW,* and *MW*) are 6 bits long; this provides for up to 64 unique windows. When a window is spilled to memory, it is still necessary to maintain a reference to the data it contains. In this architecture, windows are most easily spilled to specific locations in memory called *window frames*. These frames are located in a specific region of data memory. The approach permits windows, whether spilled or not, to be referenced via a

16 bit pointer. If the upper 10 bits of the pointer are all zero, then the window is resident in the register bank in the window specified by the lower 6 bits. Otherwise, the full 16 bit value is used to form a memory (byte) address by shifting it left 6 bits.

2.1 Procedural Activation Record Stacks

The activation record stacks are represented by a list of windows (*figure 2*). Access to the top two windows is possible using the *OLD* and *CUR* pointers as described in the preceding section. To keep track of the *thread* of windows allocated to the activation record stack, an additional window, the map window (*MW*) is needed. This serves as a map, or directory, of the stack. This map, which can be housed in any register window, contains a sequential list of six bit window pointers identifying the register windows in the stack and defining their order. The return address value for each nested procedure call is stored here as well. There is also a flag field, which indicates that the window has overflowed into memory. The *MWI* (map window index) is used to index this window. If a procedure call depth greater than 15 exists in one task, the hardware will trap and create a link for an additional map window. One register (the link) is reserved to accommodate the doubly linked list (two 16 bit pointers to either memory resident window frames, or to another register window).

The *MW* and *MWI* are altered by any *CALL* or *RETURN* instruction. The following actions are taken on each procedure call.

```
CALL:   MWI ← MWI + 1 [**]
        MW[MWI](2:25) ← PC
        MW[MWI](26:31) ← PSW2.OLD
        PSW2.OLD ← PSW2.CUR
        PSW2.CUR ← free list top [*]
        PC ← Address of Call;
```

The [*] indicates that an exception may be generated if there are no windows available to be allocated. The [**] indicates that an exception will be generated if a new *MW* must be allocated. While there appears to be quite a bit of activity, this process will be completed in one cycle, if an exception does not occur (in which case it requires three cycles, unless there is a need to overflow to memory). The *RETURN* instruction perform the reverse action.

If it becomes necessary to overflow a window from a particular activation stack into memory, the *MW* plays an integral role. Windows are copied into *window frames*, 16 word blocks in low memory, addressable using 16 bit addresses. To start an overflow, a memory resident map window image, is allocated, and a pointer to it saved in the *downward* link of the *MW*. We will call this allocated *window frame* the *memory map*. The previous contents of the downward link of the *MW* is copied to the *downward* link of the *memory map*. Next, the activation record window to be spilled is copied into a second memory window frame and the 16 bit pointer to this window frame is saved in the *memory map*. The reference in the *MW* is marked as copied. When all 15 windows of this *MW* have been overflowed the *MW* itself can be spilled. Since each *MW* contains a pair of 16 bit pointers, the *MW* linkage can be maintained regardless of whether the *MWs* are in memory or registers. When this procedure chain is reactivated, the windows must be copied back to register windows before control is transferred to them. Only those procedures that are about to be reactivated need be copied back.

If access is required to a variable down the procedure chain this map (whether resident in memory or in register windows) can be used to traverse the chain.

2.2 Traditional Stacks and Queues

To facilitate communication in multitasking systems the threaded register windows can be reconfigured as queues or traditional push/pop stacks. The two *SQW* and *SQWI* fields are used to support these structures. Four special instructions alter these pointers; *POP, PUSH, ENQUEUE* and *DEQUEUE*. All four instructions add or remove the single word elements from the *stack/queue* which is pointed to by the *SQW* plus *SQWI*. The *SQWI* is adjusted accordingly after each instruction. When 15 elements have been pushed or queued, a software trap will link another window to the *stack/queue*.

The expandability of these structures, as illustrated in *figure 3*, is especially beneficial to queue management. In most conventional systems, queues are implemented in statically allocated storage and have a circular behavior. This requires that the memory allocated be sufficiently large to accommodate the entire queue, and that base and bound checks be done. Using the threaded register window concept the queue is register resident and occupies only the minimum number of windows needed to hold the current queue contents. As the queue grows and shrinks, register windows are automatically allocated and deallocated. No checks for base/bounds need be performed.

The two *SQWs* allow access to both the head and the tail of a queue. However, frequently the entity which queues and the entity which dequeues are separate tasks. In *figure 3* two tasks are accessing the same queue. The source task uses *SQW2* to reference the queue's tail. The receiving task uses *SQW1* to reference the queue's head. The choice of using *SQW1* or *SQW2* is arbitrary, and can be left to programming convention.

The queue empty condition or stack empty condition is not automatically determined by the hardware but requires software support. However, there is a hardware

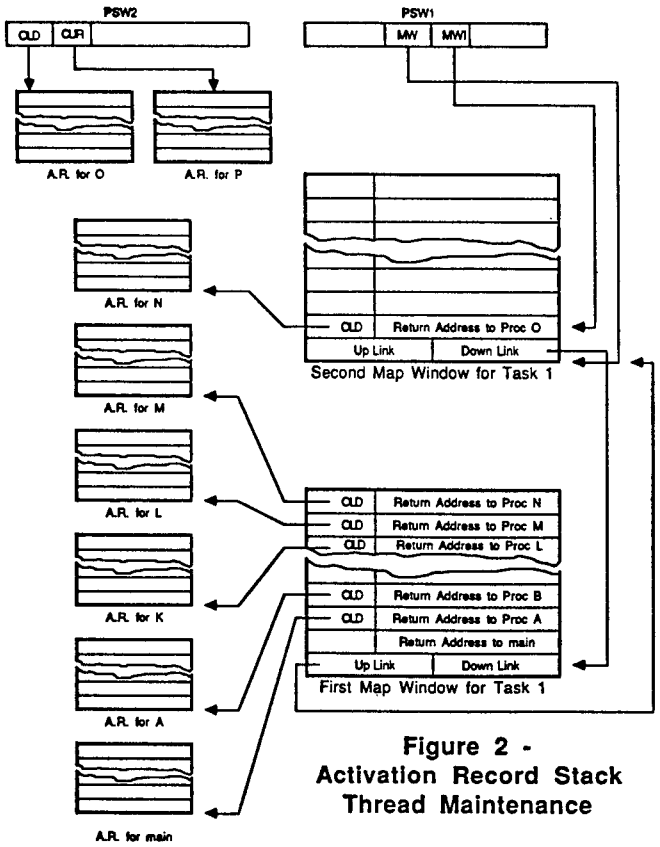


Figure 2 - Activation Record Stack Thread Maintenance

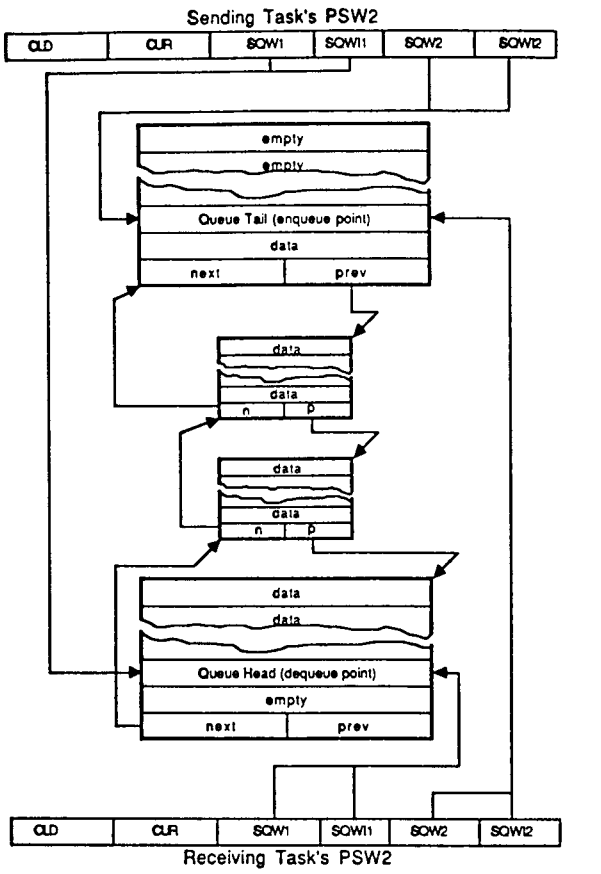


Figure 3 - Queue Window Management

aid provided. If during the execution of one of the four instructions *SQW1 = SQW2* and *SQWI1 = SQWI2* condition codes will be set.

Parts of the stack/queue can be overflowed to the memory window frames. In this case the links of its neighbor windows are changed to point to the corresponding *memory frame* address. As with other operations presented here, this set of stack/queue operations is designed to hide as much of the register window granularity from the applications programs as possible, without paying a large performance penalty.

2.3 Task Switching/Interrupt Facilities

Most of the task state can be accessed through the two PSWs. Therefore, simply saving these two words records the configuration of the activation stack, and all other window-based data structures. If these two words were saved in registers, saving the current state would require two register-to-register moves, plus any additional logic needed to determine the storage location.

In order to optimize a task context switch, any saved scheduler data, such as the location of task control blocks, scheduling queues, etc. should be made available as soon as the scheduler is passed control. To facilitate this, the architecture allows for a window to be pre-allocated for the scheduler's task *CUR* window. This window may be selected at system start-up, and a pointer to it saved in one of a set of special register called *Interrupt Control Registers (ICR)*. There are 16 of these special registers, eight are reserved for hardware interrupt vectors. The other eight may be used for software traps. The *ICR* structure is shown in *figure 4*. The window pointer field permits the static allocation of a window for use by each interrupt handler. This window remains allocated even after the handler returns. There are many advantages to this static allocation. First, it reduces the chances of a window underflow exception occurring at the time of an interrupt. Secondly, the static nature of the window makes it nicely suited to serving as a buffer for interrupts involving data transfer.

The *CALLI* instruction is used for software traps. This instruction is similar to the *CALL* instruction ex-

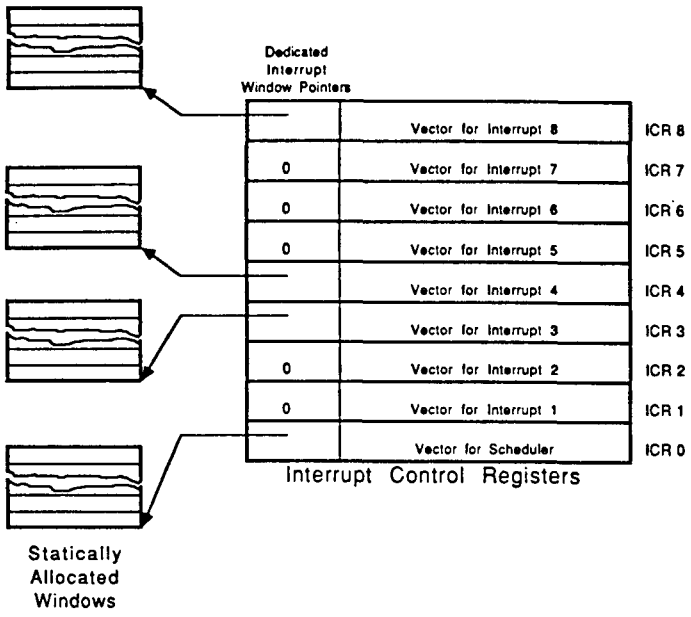


Figure 4 - Interrupt Control Registers

cept it may, or may not, select a window from the free list to be used as the new *CUR* window. *CALLI* checks the *ICR* window field. If it is non-zero, then the pre-allocated window is moved to *CUR*. A value of zero in this field causes the trap to be allocated a new window, just as in a procedure call. This same process is used to service hardware interrupts.

By using an instruction similar to a procedure call to transfer control to the scheduler, the PC and the *OLD* field of the currently active task is saved in that task's *MW*. Because of the symmetry of *CALLI* and *CALL*, once the selected task's PSWs have been loaded, the scheduler need only execute a return instruction. The *RETURNI* instruction is provided for this. It is identical to *RETURN* except it does not return the window pointed to by *CUR* to the free list, unless the *CUR* entry in the interrupt's *ICR window field* contains a zero.

2.4 Globals-Packages

Much of the emphasis of the literature on window-based RISC has been on the use of windows to solve the problems of locals and parameters in procedures. While many such architectures provide a global window, few specific proposals have been made for its usage. Also, the special needs of multi-tasking systems for global storage have not been widely addressed.

In a multi-tasking system, the global window would be used most efficiently for storage which is global to all tasks, such as scheduler information and constants. However this architecture provides another implicit window reference, the *OW* or *object window* which can be used to hold task specific global data. This window pointer in the PSW is not modified by a procedure call but it is changed when the PSW is reloaded in a task switch. It can therefore be considered to be a procedure global, task local reference. This window can be used to hold frequently accessed task specific globals. Having globals in registers was shown to be very beneficial in more conventional architectures [17]. With the added capability of task local/procedure global storage, the threaded window approach could yield highly efficient use of registers.

2.5 Register Spill Management

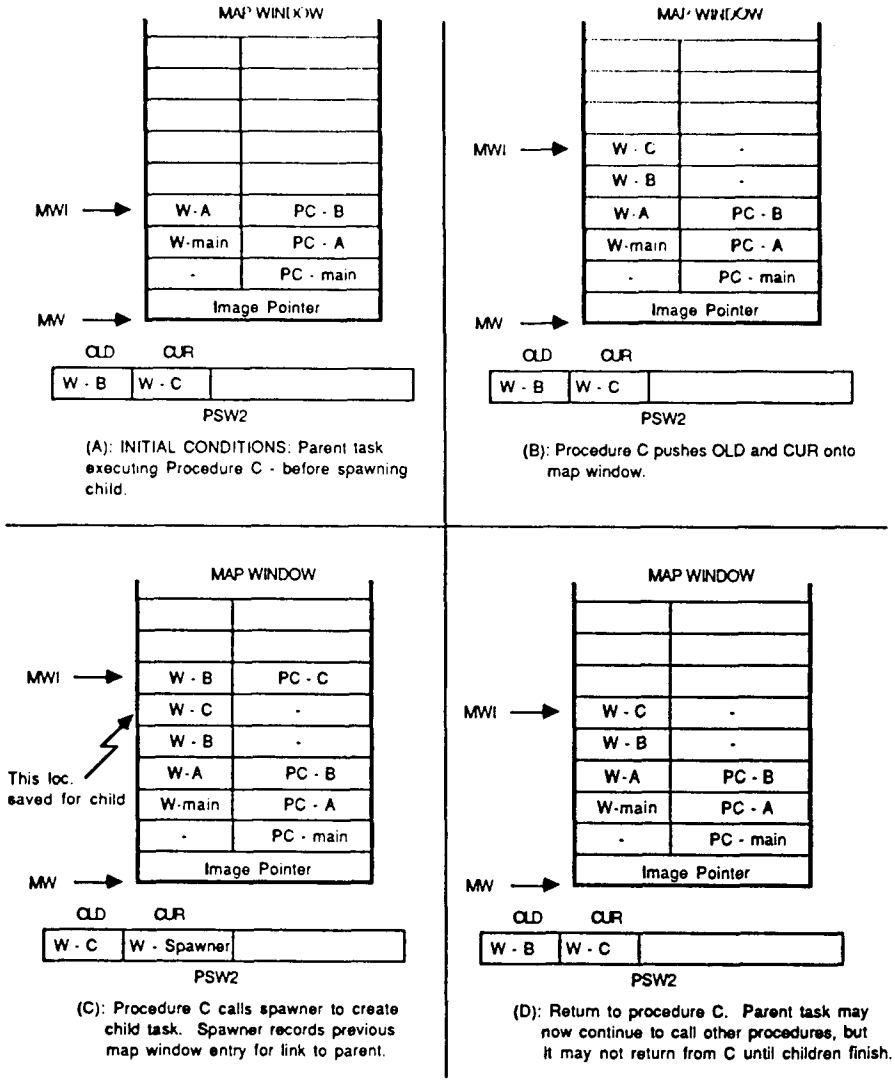
Although it is hoped that little register spillage will occur (since we are primarily storing shortlived, dynamically allocated, data in the register windows) performance gains achieved through accessing data in registers instead of in memory could easily be destroyed if efficient means do not exist for the movement of data between registers and memory. A process which uses task state data to deduce which register windows are the best candidates for spillage to memory should exist, and might run continuously as a background task.

There exists a certain implicit hierarchy of allocated windows which such a task could use to anticipate the relative likelihood that different windows will need to be accessed soon. At the top of the hierarchy are the global window and the pre-allocated interrupt and scheduler windows, which are not eligible for spillage. Next would be the Map Windows which should not be spilled until all of the windows that they point to are spilled. At the bottom of the hierarchy are the stack/queue windows and the activation record stack windows, which exhibit the most dynamic behavior.

As can be inferred from the tree-like nature of the hierarchy, there will probably be more windows allocated which fall into the lower end of the hierarchy. A system whose registers are being efficiently managed would spill these lower level windows first, keeping only a few that were recently referenced by the currently active task. The higher level windows might be spilled for tasks which are of low priority and/or have not run in a long time. This information (task priority, task dynamic history) as well as information about procedure call/return history and enqueue/dequeue history could all be used to attempt to achieve optimum register window usage.

Some concern must be directed to a window pointed to by the saved PSWs. These pointers sets up an alias. The handling of this alias should be tailored to the use of such a window. This implies a technique which is language specific.

Figure 5
Task Spawning with Cactus Stack



3 Multitasking Applications - Ada

The goal of this project was to provide an architecture capable of handling multi-tasking efficiently. To show the utility of this approach we attempted to implement some of the more complex constructs in the multitasking language Ada. Ada implementations have frequently been criticized for having too high an overhead for tasking constructs. Our main goal was to create a system where a rendezvous cost little more than a procedure call. Of course the architecture is not restricted to Ada. It should work well for any language which features multiple lines of control, or which allows multiple states to exist concurrently.

Ada tasking is characterized by synchronous communications controlled by the rendezvous, and a hierarchically shared memory, frequently referred to as the *cactus stack*. Queues are used to transfer parameters between communicating tasks. All of these constructs can be easily implemented using the threaded register window concept.

3.1 Cactus Stack

Tasking in Ada implies the concept of one task (child) being within the scope of another (parent). All data which is visible to the parent must also be visible to the child, even if that data is local to the parent. This data structure, the *cactus stack*, extends naturally from the scoping provided by the procedure *Map Window*. However, a little massaging is necessary. The manipulation of the map window during this process is shown in *fig-*

ure 5. We are assuming a compiler supplied subroutine *spawner* which actually creates the child task. Prior to calling *spawner* the parent task pushes his *OLD* and *CUR* pointers onto the map window and marks them as dummies. In this way, there is a record of all the activation record windows needed by the child task. The child keeps a pointer to a handle to the parent's map. Should the parent's map window be spilled, the handle will provide a single indirect reference for all references to that entry – i.e. each child would use the handle to locate the parent's map window. Thus, the spill manager would only need to update this one indirect pointer rather than seeking out each of the child processes.

3.2 The Rendezvous

To handle the *rendezvous*, several support routines are required. There must be support to; locate *queues*; determine if a entry is open (ready and able to process a call); and a protocol for scheduling. As was mentioned earlier, saving the task's state at the time of a task context switch requires that the PSWs be saved. In addition since we may wish to remove a particular task's data from registers, what to do with the pointers to the two top activation records must be addressed, along with the pointer to the *OW* (which we used to handle task relative information, such as the location of the task control block, the task ID, etc.). We choose to handle these three windows by pushing their pointers into the *MW*. Specifically an interrupted or suspending task will be subjected to a *CALLI* instruction. This instruction will save *OLD* window pointer and the current PC in the *MW*. The scheduler will then push pointers to the other two windows (complete with the rest of the PSWs) into *MW* using the *PUSH* command. The location and index of the *MW* is stored by the scheduler for future reference. A suspended task's *MW* is shown in *figure 6.*

Queues are not included in this task relative information, since they are not task oriented but system oriented. If queues are needed after a suspension they must be re-requested from the queue manager. The queue manager stores the locations of all queues along with an indication that the entry associated with the queue is open. We use queues to pass parameters between tasks, and to hold ready tasks waiting for the processor. The steps for a simple *accept call* are as follows.

1. Locate the queue head and tail for this entry, and receive an indication if the entry is open and executable; in addition, lock the queue. This may be implemented using a *CALLI* to a queue manager. The queue manager places the head and tail into the PSW's *SQW1* and *SQW2* fields, and returns an indication of the entry's status.

2. Place the parameters onto the queue. As mentioned before, no bounds checking is required. The parameter placement usually only requires register to register moves to the register resident *accept* queue.

3. Transfer control to an intermediate procedure which will either execute the *accept*, or suspend the caller. (Control is transferred using a *CALL* instruction which saves the current PC and the *OLD* window pointer in the *MW*.)

4. The PSWs are pushed onto the *MW* (preparing for a suspend).

5. If the accept is open, the called task's *MW* is located and its PSW loaded. The *accept* executes, and parameters are dequeued.

6. If the *accept* is closed, the queue pointers are saved, and another task scheduled.

Although this is more expensive than a procedure call (in this architecture) the overhead is minimal and involves in most cases register to register moves. Machines with small register sets usually copy all registers to memory at a procedure call. This rendezvous is less expensive than that. The time would be further reduced by using the Haberman Nassi [9] optimization (if appropriate), which could be implemented easily.

4 Conclusion

Register Window Threads are capable of handling the dynamically created data of multiple tasks in on-chip storage. This ability enables tasking languages such as Ada, Occam, and Modula to benefit from RISC technologies which use large on-board memories. This is of special importance since these languages encourage programmers to use many centers of control and frequent context switches. These languages are often suggested for use in real-time applications, if task context switches cannot be executed efficiently, they will not be effective.

We believe that the same features of the architecture which supports multitasking can also be used to save the environments of multiple object states, as is found in object oriented languages such as Smalltalk. This is due to the fact that tasking languages and object oriented languages have one similarity. In both types of languages control transfers from one environment to another frequently. This control transfer should be optimised, just as procedure control transfer has been optimised in block structured languages such as C and Pascal. In addition these languages should also be able to execute a procedure call with the same low overhead as block structured programs.

The technique developed here offers these capabilities and also allows communication queues and traditional push/pop stacks to reside in registers; thus extending the benefits of the activation record stack

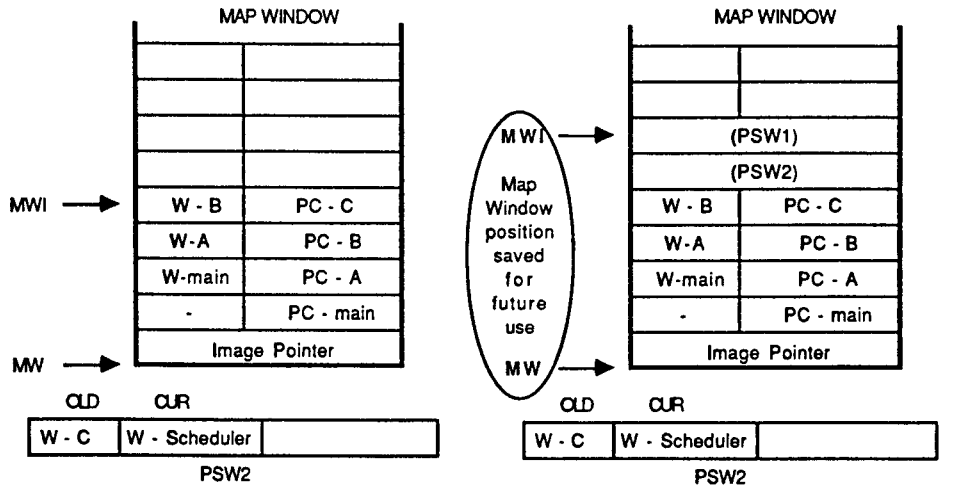


Figure 6 - Task Suspension

to these classes of shortlived heavily access dynamically allocated data. Our technique differs from other techniques that support multitasking in two important ways. Other techniques segment the on-board storage. This means that the number and size of on-chip environments is statically set. Our system dynamically creates, and sizes, new environments. In our system, any number of environments may exist at one time, and this number may vary freely.

The development of this approach is still in its formative stages. A simulator and compiler for the proposed machine are currently under development. The completion of these projects should permit extensive quantitative measurements of the effectiveness of this architecture. At this time, we feel that it is important to explore the degree to which the register window concept can be exploited to make the organization of on-chip memory banks more general.

References

[1] *AM29000 User's Manual.* Advanced Micro Devices, 1987.

[2] A. V. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1975.

[3] Robert P. Colwell, Charles Y. Hitchcock, E. Douglas Jensen, H. M. Brinkley Sprunt, and Charles P. Kollar. Computers, Complexity and Controversy. *Computer*, September 1985.

[4] D. R. Ditzel and J. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, 1982.

[5] R. Eickemeyer and J. Patel. Performance Evaluation of Multiple Register Sets. In *Proc. 14th An-*

nual International Symposium on Computer Architecture, 1987.

[6] R. Eickemeyer and J. Patel. Performance Evaluation of On-chip Register and Cache Organization. In *Proc. 15th Annual International Symposium on Computer Architecture*, 1988.

[7] E. G. Gehringer and R. P. Colwell. Fast Object-Oriented Procedure Calls. *Computer Architecture News*, 14(2), 1986.

[8] J. Goodman and W. D. Hsu. On the Use of Registers vs. Cache to Minimize Memory Traffic. In *Proc. 14th Annual International Symposium on Computer Architecure*, 1986.

[9] A. N. Habermann and I. R. Nassi. *Efficient Implementation of Ada Tasks.* Technical Report CMU-CS-80-103, Carnegie Mellon University, 1980.

[10] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: a VLSI Processor Architecture. In *Proc. CMU Conference on VLSI Systems and Computations*, pages 337–346, 1981.

[11] D. A. Patterson and C. H. Sequin. A VLSI RISC. *Computer*, 15(9), 1982.

[12] F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. Kahn, K. K. Lai, and J. R. Rattner. Supporting Ada Memory Management in the iAPX-432. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, 1982.

[13] D. Quammen, D. R. Miller, and D. Tabak. Register Window Management for a Real-Time Multitasking RISC. In *Proc. of the 22nd Hawaii International Conference on System Sciences*, 1989.

[14] G. Radin. The 801 Minicomputer. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, 1982.

[15] Daniel Tabak. *RISC Architecture.* Research Studies Press, 1987.

[16] G. Taylor, P. Hilfinger, J. Larus, and D. Patterson. Evalutation of the SPUR Lisp Architecture. In *Proc. 14th Annual International Symposium on Computer Architecture*, 1986.

[17] D. W. Wall. Register Windows vs. Register Allocation. *SIGPLAN*, 23(7), 1988.